DPT 050N 105
PRICE 19.95
M STR 50 4570

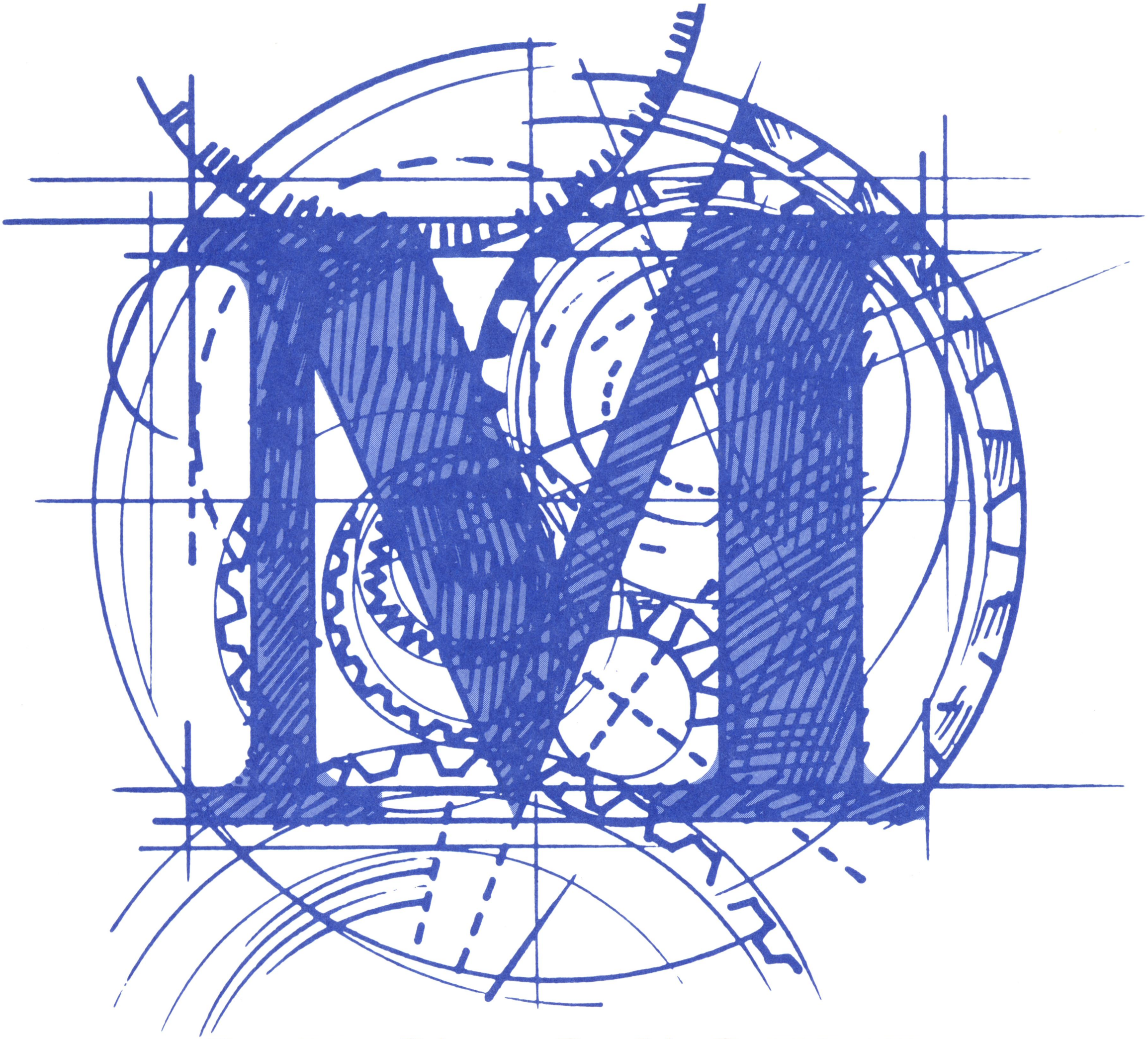
MICROSOFT® MACINATIONS

An Introduction to Microsoft® BASIC For the Apple® Macintosh™

The Waite Group · Mitchell Waite · Robert Lafore · Ira Lansing

MICROSOFT® PRESS

MICROSOFT®
MACINATIONS



MICROSOFT® MACINATIONS

An Introduction to
Microsoft® BASIC
For the Apple® Macintosh™

The Waite Group · Mitchell Waite · Robert Lafore · Ira Lansing

MICROSOFT® PRESS

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
10700 Northup Way, Box 97200, Bellevue, Washington 98009

Copyright © 1985 by Mitchell Waite
All rights reserved. No part of the contents of this book may
be reproduced or transmitted in any form or by any means without
the written permission of the publisher.

Library of Congress Cataloging in Publication Data
Waite, Mitchell.
Microsoft macinations.
Includes index.
1. Macintosh (Computer)—Programming.
2. Basic (Computer program language).
I. Lafore, Robert (Robert W.) II. Lansing, Ira.
III. Waite Group. IV. Title.
QA76.8.M3W35 1985 005.2′65 85-13863
ISBN 0-914845-34-9

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 FGFG 8 9 0 9 8 7 6 5

Distributed to the book trade in the
United States by Harper and Row.

Distributed to the book trade in
Canada by General Publishing Company, Ltd.

Distributed to the book trade outside the
United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging in Publication Data available

Apple® is a registered trademark of, MacPaint™ and MacWrite™ are trademarks of,
and Macintosh™ is a trademark licensed to Apple Computer, Incorporated.

IBM® is a registered trademark and PC/XT™ is a trademark of
International Business Machines Corporation.

Microsoft® is a registered trademark of Microsoft Corporation.

This book is dedicated to
Steve Jobs and Bill Gates,
whose combined visions made
this book possible.

CONTENTS

Acknowledgments ix

Introduction xi

SECTION I Introduction to BASIC

1
BASIC Preliminaries 3

2
Instant BASIC 27

3
Programs: Creating, Editing, and Saving 45

4
Introducing Graphics 71

5
Variables and INPUT 95

6
Loops 117

7
Decisions 143

8
Subroutines and Program Organization 159

9
Arrays and Read-Data 177

10
Strings 201

SECTION II Advanced BASIC for the Macintosh

11
The Mouse and Event Trapping 229

12
Menus 251

13
Buttons 279

14
Windows, Dialog Boxes, and Edit Fields 303

15
QuickDraw 339

16
Files 389

17
Animation 431

18
Multivoice Sound 459

Index 489

ACKNOWLEDGMENTS

The authors would like to thank the following individuals for helping to make this book possible. At The Waite Group we thank Michael Pardee for his contributions to the subjects of menus, buttons, windows, and files, and Paul Hoffman for his excellent sound chapter. We bless the wonderful person at TWG who supplied the great chocolate chip cookies during the rainy Christmas weeks Robert and Mitchell spent writing. A wink and kind nod to Salley Oberlin and the folks at Microsoft Press who took the authors' creativity and tamed it into a consistent and unified whole. Barry Preppernau and Chris Matthews provided excellent technical input. Special kudos to Chris for his great program contributions. A wave and smile to Nahum Stiskin who brought The Waite Group to Microsoft Press in the first place. A final bow and *domo arigato* goes to David Laraway at Microsoft Press, who took our final manuscript and made its presentation crisp, succinct, and trimmed of unnecessary verbiage.

INTRODUCTION

The Macintosh computer is justly famous for its revolutionary "user interface." This term simply means the way in which we humans interact with the computer and harness its power. The Macintosh is one of the most powerful personal computers ever built; at the same time, its artful blend of pull-down menus, movable windows, dialog boxes and buttons, and the well-known mouse makes the Macintosh easy to use. It is this ease of use that makes the Macintosh so unique, and so appreciated by those who work and play with it.

Such a multidimensional machine deserves a programming language that makes use of its many "bells and whistles." Because beyond its basic design, a computer—even the Macintosh—is only as good as the *software* programs that make it "think." Without good software, one wit has observed, a computer is nothing but a very expensive paperweight. And software is, to a large extent, a function of the programming *language* used to write it. It must be capable of using the powerful features of the computer, but at the same time, like the computer itself, it must be as easy to use as possible.

You will find that Microsoft BASIC 2.0 for the Macintosh meets the twin requirements of power and simplicity. This comprehensive, two-part book will help you harness that power by showing you how to write programs that will make the Macintosh really perform.

The first half of the book teaches the fundamentals of using the Microsoft BASIC language, including such concepts as variables, graphics, loops, decision making, subroutines, strings, and arrays. Although these ideas may sound strange now, in no time at all you'll be bandying them about with ease.

In the second half you will learn how to write programs that use menus, windows, and buttons—the elements of Mac's famous user interface. You will learn about "event" programming—a modern method for programming responses to the outside world. Windows and how the information in them is updated when one overlaps another are both clearly explained. If you have heard of QuickDraw, Macintosh's built-in graphics routines, you'll learn everything you need to know about them here. We teach you how to use the graphics pen, how to plot points, draw lines, arcs, rectangles, ovals, and complex designs and patterns. There are also chapters on disk files, graphics animation, multi-channel sound, and music.

Many of the example programs used in this book are useful and interesting in their own right. For instance, a pattern-maker program gives you a FatBits-like interface that lets you design any custom pattern you want and use it to make a background or fill a shape. Another program presents a race between two rabbits to demonstrate the principles of animation. A sound-lab program lets you create and edit custom waveforms and then listen to them. You can use it to design special-quality sound effects, musical instruments, and so on. There is even a complete rendition of a four-voice Bach concerto.

IS THIS BOOK FOR YOU?

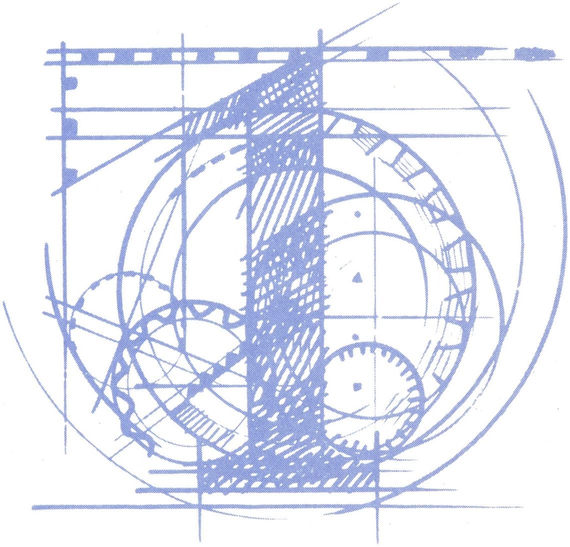
We've written this book with both the new and the experienced programmer in mind. If you're a beginner, this book will get you started. We begin with the most fundamental principles of BASIC programming and build on them to more complicated topics. We believe that a person best learns how to use a programming language by being *shown* how it works, rather than simply being *told* how it works in abstract terms. Accordingly, we focus on example programs that we encourage you to type in as you go along. We support the programs with figures that show their actual output, so you can compare your results with ours and know if you're on the right track. In fact, even if you don't have access to a Macintosh (either a 128K or 512K version) you'll still be able to learn a lot about how Microsoft BASIC for the Macintosh works by reading the explanations and studying the programs and figures.

Whether you're a new programmer or an old hand with code, this book won't outlive its usefulness after you've learned the basics about Microsoft BASIC for the Macintosh. We're confident that it will become a valued reference source as you expand your programming proficiency. We hope that you keep it near your computer and turn to it when you need to refresh your memory or expand your knowledge of BASIC.

The Macintosh is one of the most exciting personal computers around today, and Microsoft BASIC is one of the most exciting programming languages available for it. We've written this book to help you get the most out of both. So turn the page and let's begin!

SECTION I

Introduction to BASIC



BASIC Preliminaries

This chapter is the first stage of our journey into learning Microsoft BASIC. The first thing we're going to do is discover what tools Microsoft provides us with in the BASIC Interpreter package. Our next step will be to create working copies of the BASIC disk to actually use for our programming. This copy is a sort of insurance policy that protects us; if something should go wrong with the copies, we'll still have the original disk to fall back on. After that, we'll start up BASIC and orient ourselves in the BASIC desktop environment.

WHAT'S IN THE BOX?

When you first open your new Microsoft BASIC Interpreter package, you will find a 3½-inch microfloppy disk, a reference manual, and a registration card. The registration card contains your BASIC serial number. You must fill out the card and send it to Microsoft in order to be eligible for the Microsoft Help Hotline for technical support, the Microsoft Product Replacement Plan if your disk is defective, and the Microsoft Product Upgrade Plan, which enables you to buy later "new and improved" versions of BASIC at a reduced price. So, it really is worth your time to fill in the card and drop it in your mailbox.

As for the manual, you'll want to keep it close at hand because it contains definitions of how the language works and what the different statements do. Although it's not written to teach you BASIC, it will serve as an invaluable reference aid at various crossroads on your journey into learning BASIC.

The last item to take out of the package is the BASIC disk, which contains the *operating system* that enables the Macintosh to function, as well as the BASIC language files that let us program. We're going to be working with this disk at first, so let's learn a little bit about it now.

The BASIC disk, like all other disks used with the Macintosh, is 3½ inches square. Its official title is Microsoft BASIC Interpreter for Apple Macintosh. There is a *serial number* on the disk that corresponds to the number on your registration card.

If you already own a Macintosh, you are probably familiar with the MacWrite and MacPaint disks. To use MacWrite, you have to insert the disk containing the MacWrite software. The same is true of Microsoft BASIC: You have to insert the disk containing Microsoft BASIC in the Macintosh if you want to program in BASIC. Without this disk, you cannot write a BASIC program and will find it difficult to learn how to program.

Write-protecting the Microsoft BASIC disk

Computers are wonderful because, like "man's best friend," they rarely complain, are dependable, and will eventually even fetch the morning newspaper. But also like man's best friend, computer equipment sometimes does things we don't expect and gets us into trouble. For example, although a microfloppy disk is very reliable, sometimes excessive heat will render its contents unreadable. Or like the song says, sometimes "smoke gets in its eyes" and gums up the disk's magnetic surface. No matter how careful we are, these things *may* happen. So what to do?

To protect ourselves from possible foul-ups like this, we need to *back up* our original BASIC disk by making copies of it to work with. That way, if a copy is damaged we always have the pristine original to recopy.

We're going to go through the steps of backing up our original and making working copies of it very soon. But first, there is one physical method of protecting our original disk that we're going to take care of before we even put it in the machine for copying. If you hold up the disk with the label facing you, you will notice a tiny window in the upper right corner. This is called the *write-protect window.* If the window is closed (you can't see through it), the Macintosh can both *read* information from the disk and *write* (or record) information on the disk. If the window is open, the Mac can read information from the disk but *not* write to it—the disk is therefore *write protected.* We can open or close the window by sliding the *write-protect tab* in its track behind the window.

If something should go wrong with the Macintosh and the disk we are working with is not write protected, the Mac could try to store bad information on the new disk and possibly ruin it. If the disk is write protected, there are circuits in the Mac that will abort any attempts to record information—good or bad—on the disk. Since we can still copy our original BASIC disk if it's write protected, let's slide the tab open on the disk right now. That way, if something does go awry while we're copying the disk, we won't destroy it.

Now that we've protected our disk, we don't have to feel nervous about working with it. So let's plunge on and see where the next step of the journey leads us.

THE MICROSOFT BASIC DISK WINDOW

When you turn on the Macintosh and insert your new BASIC disk for the first time, the disk drive whirls for a few seconds and then you're presented with a *window* on the Macintosh *desktop*, as shown in Figure 1-1. (If you don't see this window, the *disk icon* labeled Microsoft BASIC in the upper right corner will be black; to open the window, place the arrow pointer on the disk icon and quickly *double-click* the mouse button.)

What is a window? In general, it's a graphic area on the desktop. What it's used for, what's displayed inside it, and what happens when we type in it depends on the program we're using. In this case, the purpose of the window is to show us the contents of the BASIC disk—the programs that are stored on it.

The programs are displayed in the open window as *icons*. Notice the two icons in the window, one labeled Microsoft BASIC (b) and marked with the Greek letter π, and the other labeled Microsoft BASIC (d) and marked with a dollar sign. Each icon represents a different version of BASIC: BASIC (b) is the *binary* version, and BASIC (d) is the *decimal* version. The primary difference between the two versions is the way they round off numbers. The decimal version uses the BCD (short for *binary coded decimal*) format for calculating numbers. The binary version uses a

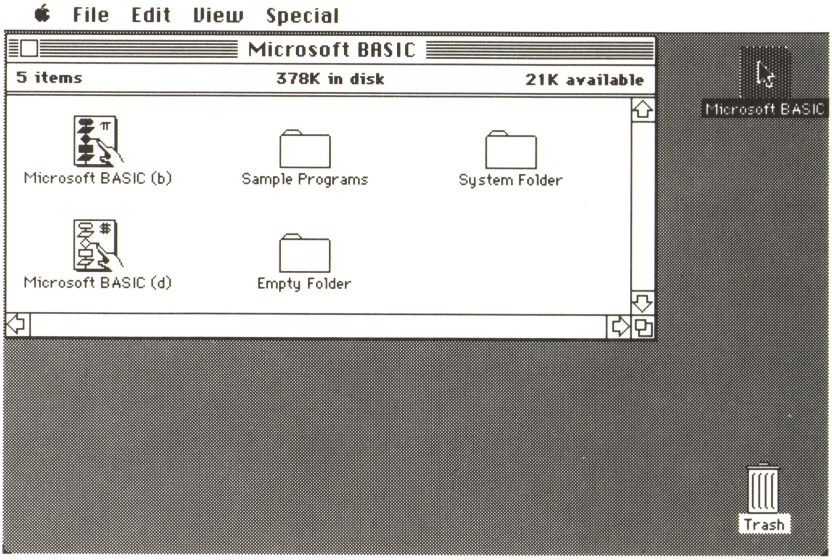

File
Edit
View
Special
Microsoft BASIC
5 items
378K in disk
21K available
Microsoft BASIC (b)
Sample Programs
System Folder
Microsoft BASIC (d)
Empty Folder
Microsoft BASIC
Trash

FIGURE 1-1

The BASIC disk window

format known as IEEE, which is an acronym for the Institute of Electrical and Electronic Engineers—a professional organization that establishes industry standards.

The decimal version of BASIC avoids *round-off errors* when writing numbers involving dollars and cents. Round-off errors result from inaccurate rounding of the results of repeated calculations, and even though the round-off error might be quite small for one mathematical operation, errors can really add up fast if you're doing thousands or even hundreds of thousands of repeated calculations. This becomes critically important in the financial world, where a repeated error of only a few cents can amount to losses of millions of dollars. So if you eventually want to write business and financial applications, this is the version to use. Also, if you do happen to acquire programs that were written using Microsoft's 1.0 release of BASIC for the Macintosh, they will run with the decimal version.

Although the decimal version is best for working with money numbers, the binary version is the one to call on when speed is important, such as with animated graphics. Since the binary version is faster, and since most of the programs we create don't need the mathematical ability of the decimal version, we use the binary version almost exclusively for the programs we create in this book.

Now that we've had a look at the Microsoft BASIC disk window and discussed the differences between the two versions, it's time that we made working copies of BASIC to use from now on. There are two reasons for this. We've already mentioned the first: Making a backup is a wise safety precaution. The second reason is that if we *dedicate* one working disk for the binary version of BASIC and one for the decimal version, we can erase the version we don't need on each disk, which leaves us a lot more room to save the programs we create ourselves. This is especially important when using a single-drive Macintosh, as it lets us avoid a lot of the disk-swapping that becomes necessary if we try to save programs on a disk that's almost full.

So, in the following sections we're going to make a copy of the entire Microsoft BASIC disk, then put our original away and use this "backup disk" to make two more copies, one with the binary version of BASIC and one with the decimal. Let's get started.

BACKING UP OUR MICROSOFT BASIC DISK

We are going to need three blank disks to complete our backup procedures: one for the master backup, and one each for the binary and decimal versions. We will consider the disk we're copying from the *source* disk, and the one we're copying to the *destination* disk. In the following sections, we'll give disk-backup instructions for both single-drive and dual-drive Macintosh systems.

Copying the Microsoft BASIC disk on a single-drive Macintosh is easy, but we do have to "swap" disks in and out of the drive several times. This swapping is required because the Mac copies a "chunk" of information off the source disk into its random access memory (RAM), then copies this chunk from memory onto the destination disk. It repeats this process until the entire disk has been copied. Since the memory can't hold the entire contents of a disk at one time (unless we have a 512K Mac), we are prompted by the Mac to remove the source disk when memory gets filled and insert the destination disk, then remove the destination disk and insert the source, and so on.

The first thing we need to do is close the disk window by choosing Close All from the File menu, like this:

Point:	to the File menu by moving the mouse pointer to the word File in the menu bar at the top of the screen
Press:	the mouse button and hold it down
Drag:	the pointer slowly down the menu while holding down the mouse button

Each of the menu items becomes highlighted, or selected, as the pointer passes over it.

Stop:	when the Close All command is highlighted

At this point, your Mac's screen should look like Figure 1-2.

Release:	the mouse button with Close All selected

When we release the mouse button with Close All selected, the disk window shrinks into the Microsoft BASIC icon, turning the icon black. We say that the window is now *closed.* If we were to pull the File menu down again and drag the pointer to Open, the disk window would *open* and appear back where it was before we closed it. This process of pointing to a menu, holding down the mouse button, dragging the pointer to an

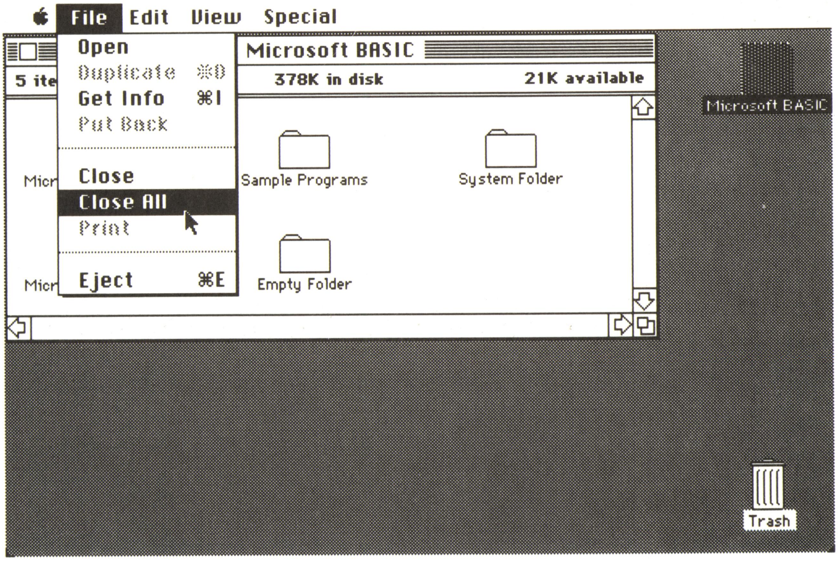

File
Edit
View
Special
Open
Duplicate ⌘D
Get Info ⌘I
Put Back
Close
Close All
Print
Eject ⌘E
Microsoft BASIC
5 ite
378K in disk
21K available
Sample Programs
System Folder
Empty Folder
Micr
Micr
Microsoft BASIC
Trash

FIGURE 1-2

Choosing Close All from the File menu

item, and releasing the button is called *choosing* from a menu. From now on when we want you to choose from a menu we'll say it like this:

Choose: Close All from the File menu

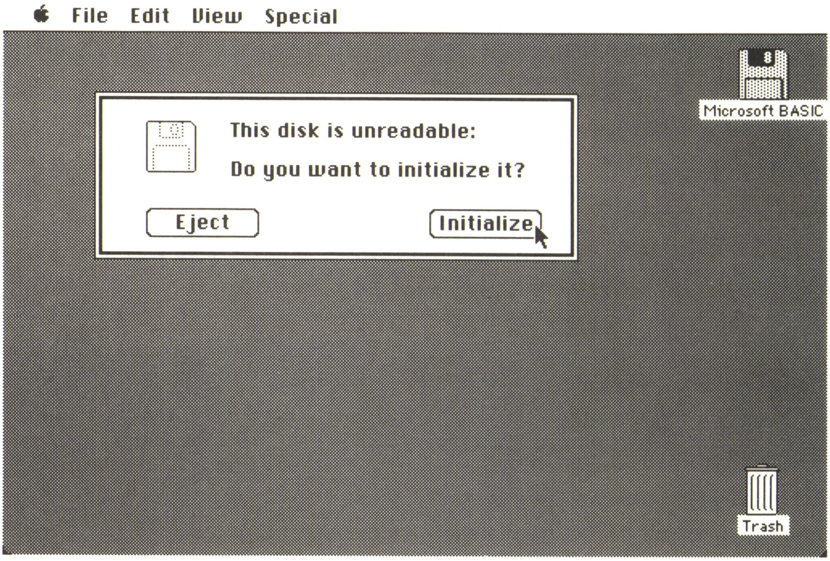
Now that the disk window is closed we are almost ready to make our backup disk. But first, we need to get a blank destination disk ready to receive programs because a blank disk comes "unformatted" and the Macintosh is unable to store information on an unformatted disk. If you're using a *single-drive* Mac:

Choose: Eject from the File menu

The disk drive whirls for a moment, and then the BASIC disk pops out. The icon for the ejected disk becomes shaded to indicate that the disk is no longer in the drive.

Initializing the blank disk

Now, insert the blank disk into the Macintosh drive (if you have an external drive, insert the blank disk in it without ejecting the BASIC disk from the internal drive). The Macintosh recognizes that this disk has not been



This disk is unreadable:

Do you want to initialize it?

FIGURE 1-3

The initialization dialog box

used and responds with the *dialog box* shown in Figure 1-3. A dialog box usually asks us to respond to offered choices by clicking buttons, or to provide information by typing into a text window. The dialog box shown in Figure 1-3 provides us with two buttons, Eject and Initialize.

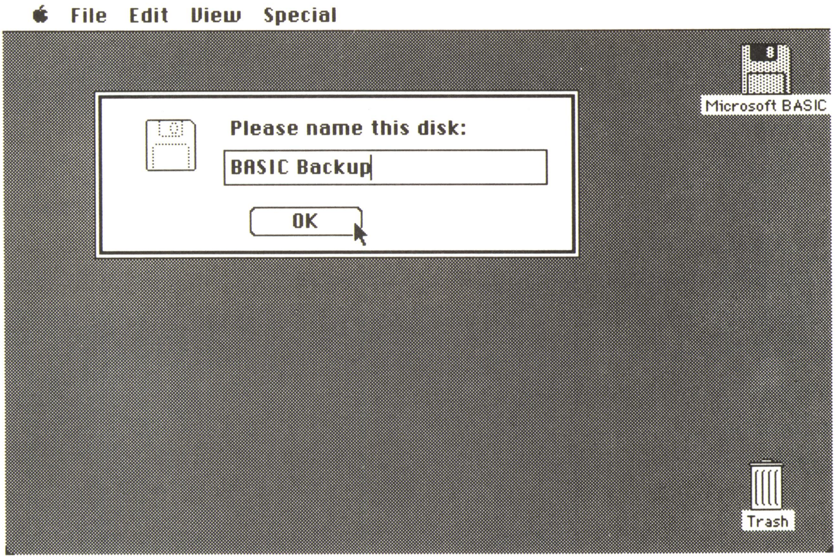
The term *initialize* describes the process by which the Macintosh records certain organization and tracking information on the magnetic surface of the disk so that programs can be stored on it. You do not need to understand how this works until we get into disk file access from BASIC, so for now just remember that any blank disk needs to be initialized.

Click: on the Initialize button in the dialog box

Naming the blank disk

After we click the Initialize button, the Macintosh begins spinning the disk and initializing it. Once initialization is complete, the Macintosh presents us with the dialog box shown in Figure 1-4, which asks us to name the disk. If we now type something on the keyboard, the letters appear in the *name window* of the box. Let's type *BASIC Backup.* If you make a mistake while typing, just use the Backspace key to back up over the letters, and retype the name correctly. When satisfied with the name:

Click: the OK button



Please name this disk:

BASIC Backup

FIGURE 1-4

Naming the blank disk

Copying the original disk

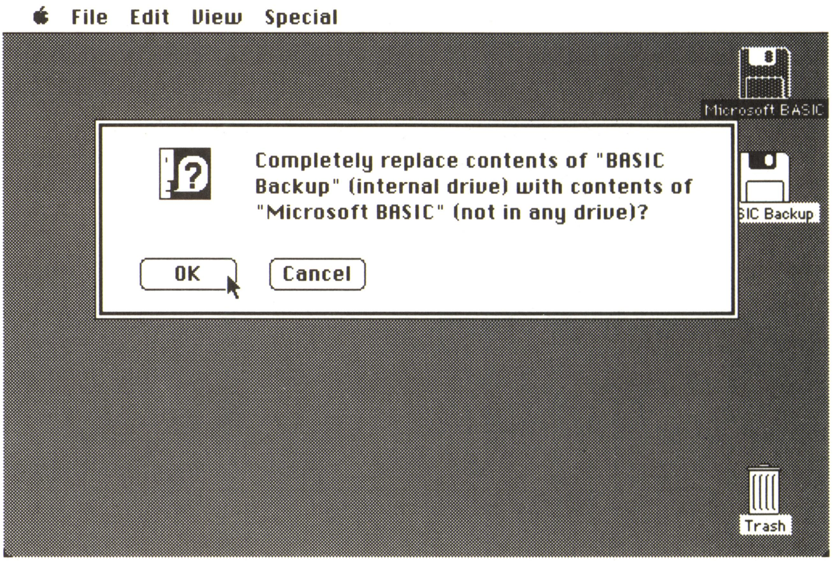
After the disk-name dialog box disappears, a new disk icon appears directly below the icon for the Microsoft BASIC disk. We are finally ready to copy Microsoft BASIC onto our freshly initialized disk.

In previous incarnations of computers we would now type some complicated command from the keyboard. But with the Macintosh, making a copy is a much more visual process. In fact, we type nothing. All we need to do is drag the icon for Microsoft BASIC on top of the BASIC Backup icon. Follow these instructions:

Point: to the Microsoft BASIC icon
Click: the mouse button and hold it down
Drag: the pointer slowly to the BASIC Backup disk while holding down the mouse button

As we drag the pointer, an *outline* of the Microsoft BASIC disk follows it. When the pointer and disk outline reach the BASIC Backup disk icon, the icon turns black, indicating that it's selected. Now:

Release: the mouse button



Completely replace contents of "BASIC Backup" (internal drive) with contents of "Microsoft BASIC" (not in any drive)?

FIGURE 1-5

The dialog box that appears after the outline of the source-disk icon has been dragged to the destination-disk icon

The Macintosh presents us with the dialog box that is shown in Figure 1-5, which asks if we really want to copy everything from one disk to the other.

Click: the OK button

If you are using a single-drive Macintosh, you will have to take turns putting the source and destination disks in the drive slot. The Macintosh will tell you exactly when to do all of this swapping, and will even automatically eject the disk it no longer wants. If you are using an external drive, you don't have to do anything but wait for a minute or two while the files are copied from the source disk in the Macintosh to the destination disk in the external drive.

During the copy procedure, a small *alert box* appears listing the number of files being copied, and the number is reduced as each file is copied. When all the copies are made, the alert box disappears and the desktop returns to normal. The BASIC Backup disk is now an exact duplicate of the Microsoft BASIC disk.

After the copying has been done, eject the original Microsoft BASIC disk and store it in a safe place, such as in its original box.

Restarting with our BASIC Backup disk

At this point there are two disk icons on the desktop, one for the original Microsoft BASIC disk and one for our BASIC Backup disk. But since we won't be working with the original BASIC disk anymore, we need to restart the Macintosh with the backup disk. To do this, we have to eject the BASIC Backup disk, turn the Mac off, turn the Mac on again, and reinsert the disk. Why? Because when we switch on the Macintosh and insert a disk containing a system folder, that disk becomes the *dominant* disk. From that point on, until we turn off its power, the Macintosh needs this disk to operate. If we try to operate the Mac with a different disk, it will constantly prompt us to insert the dominant disk. It's as if the Mac doesn't want us changing its "mind" in mid-thought. But if we switch off its power it forgets all about the disk, and we can restart with any other disk that contains a system folder. Let's do that now with our BASIC Backup disk so we can get on with our journey.

Choose:	Eject from the File menu

After a short pause, the Mac pushes the backup disk out of the drive.

Switch:	the power off, wait a second, and then turn it on again

The Mac now shows a blinking question-mark icon, meaning that it's hungry for a disk.

Insert:	the BASIC Backup disk into the disk drive

The Mac accepts the disk and soon displays the startup desktop.

Creating binary and decimal work disks

We're now ready to create our two work disks, one with the binary version of BASIC and the other with the decimal version. Doing this is a simple matter of copying the entire BASIC Backup disk to our second blank disk and putting one of the versions of BASIC in the trash can; then copying the backup disk to our third blank disk and trashing the other version. Since most of the examples in this book will be created with binary BASIC, let's go through the steps for creating a work disk containing that version.

Eject:	the BASIC Backup disk
Insert:	a blank disk

At this point we have to go through the initializing disk swaps. When the dialog box appears asking us to name the disk, let's call it BIN BASIC.

When its disk icon appears on the desktop below the BASIC Backup icon:

Drag: the outline of the BASIC Backup icon to the BIN BASIC icon

After following the prompts and swapping the two disks until the copy procedure is completed, eject the BIN BASIC disk and restart the Macintosh with it. Now we're ready to get rid of the decimal version of BASIC.

The first thing we need to do is open the disk window, so we can get at the decimal-BASIC icon. The BIN BASIC disk icon should be selected (black) at this point; if yours is not, simply click on it with the pointer. To open its window, choose Open from the File menu, or simply double-click on the disk icon.

Now that the window is open, we can drag the decimal-BASIC icon (the one with the (d) in the name) down to the Trash icon, as shown in Figure 1-6. When the Trash icon turns black, we know that it's ready to receive the file and we can release the mouse button. The Macintosh beeps at us and displays the dialog box shown in Figure 1-7, asking us if we're sure we want to erase the file. This is a precaution built into the Macintosh operating system, giving us a chance to stop and think lest we hastily trash an important application-program file by mistake. This is no mistake, so we'll click the OK button and watch the icon disappear into the trash can. At this point the file is not actually erased, but it will be if we choose Empty Trash from the Special menu, run any application program (such as one of the

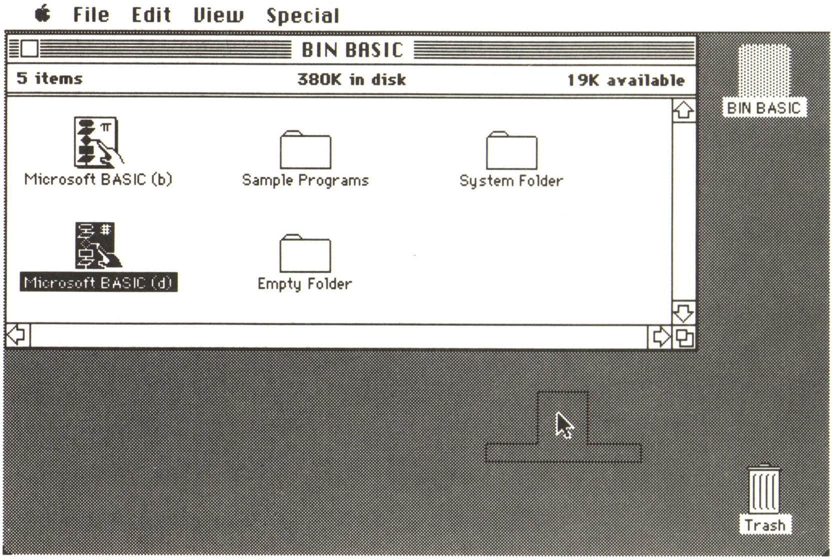
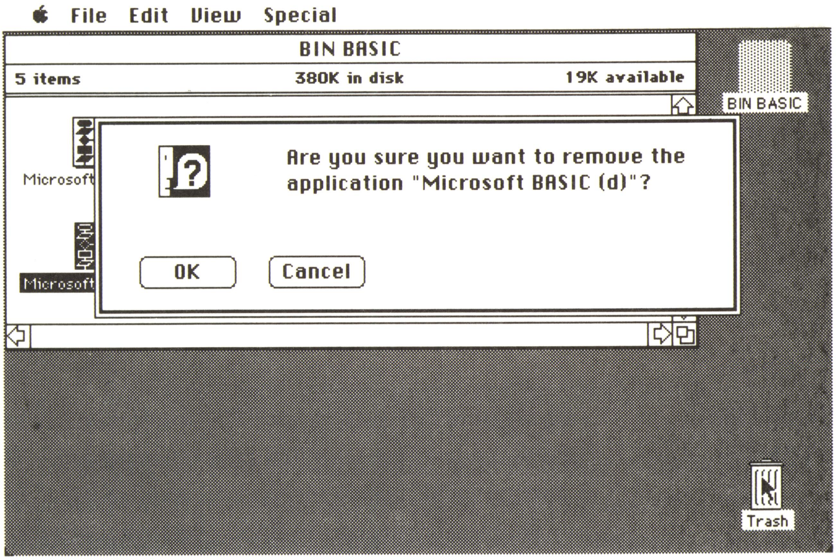

File
Edit
View
Special
BIN BASIC
5 items
380K in disk
19K available
Microsoft BASIC (b)
Sample Programs
System Folder
Microsoft BASIC (d)
Empty Folder
BIN BASIC
Trash

FIGURE 1-6

Trashing the decimal version of BASIC



Are you sure you want to remove the application "Microsoft BASIC (d)"?

FIGURE 1-7

The dialog box that appears when an application program is trashed

BASIC versions), or eject the disk. Let's choose Empty Trash now. The disk drive whirls for a couple of seconds, and the decimal version of BASIC on this disk is sent into oblivion.

We can see the practical result of trashing the decimal-BASIC file by looking at the *status line* just beneath the title bar of the window. It now shows that there is 299K of information on the disk, and 101K of space on the disk available to us for the programs we'll be creating.

The space on the disk is divided into units called *bytes.* What is a byte? It is a unit of information representing one *character,* such as a letter or punctuation symbol. A byte may not always contain a character, but in general you can think of it that way. In the case of the Macintosh, the units of storage on the disk are called *kilobytes,* abbreviated to K, which are each 1,024 bytes. A Macintosh disk has room to store about 400K (or $400 \times 1{,}024 = 409{,}600$) bytes.

Programs, folders, and icons consume different amounts of space on a disk, reducing the total number of bytes of free space available. The 101K of available space on our disk should be enough for the work we will be doing in the remaining chapters of this book, but we can give ourselves even more room by trashing the Sample Programs folder (if we ever want to run the programs contained in this folder, we can use the BASIC Backup disk). So, simply drag the folder down to the Trash icon and choose Empty Trash from the Special menu. This will leave us 161K of free space.

Now that you know how to create a work disk, go ahead on your own and make a decimal-BASIC disk (name it BCD BASIC, so we'll all know which disk to use when we need the decimal version of BASIC). When you're finished, you can put the decimal disk and the BASIC Backup disk in your disk-storage box and restart the Macintosh with the BIN BASIC disk, because that's the one we will be working with until later on in the book. Unless we tell you otherwise, from now on you can assume that whenever we talk about BASIC we mean the binary version.

STARTING BASIC

Now that we've restarted the Macintosh with our binary-BASIC work disk, we're ready to take our first look at the program. We have two options for starting BASIC: selecting the icon and choosing Open from the File menu, or double-clicking on the BASIC (b) icon. Let's do it the fast way:

Dbl-click: on the Microsoft BASIC (b) icon in the disk window

The Microsoft BASIC desktop environment now appears on the screen, as shown in Figure 1-8. Let's learn how that environment is used.

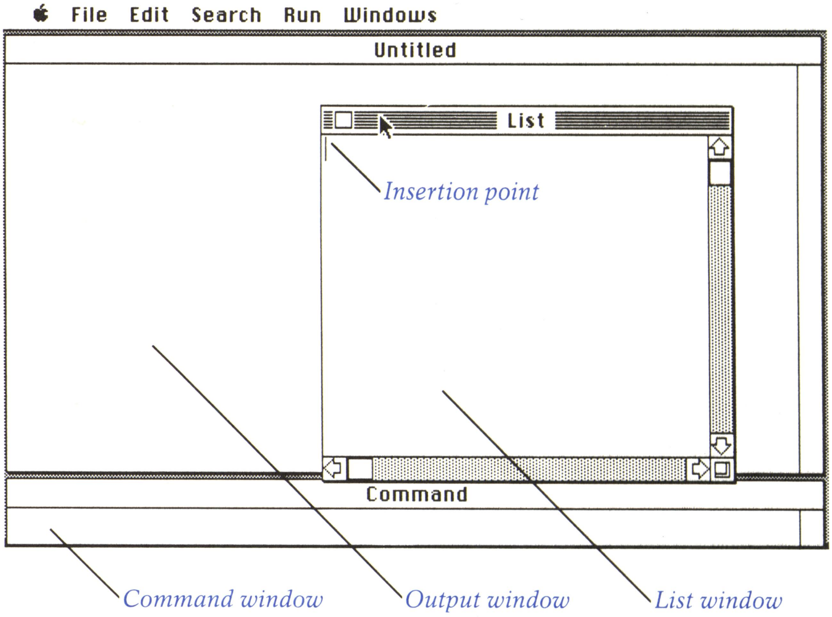


FIGURE 1-8

The Microsoft BASIC desktop and its three windows

BASIC WINDOWS

The Macintosh BASIC screen can be thought of as the top of a programmer's work desk. There are three windows on this desktop, as you can see in Figure 1-8. These are the *List* window, the *Command* window, and the *Output* window. The Output window's title bar is labeled *Untitled.* Later we will learn how to give it a different name.

You can think of each window as a piece of paper on the desktop. These "pieces of paper" each have special properties, including the ability to be made larger or smaller and to be moved horizontally and vertically on the desktop. They can also overlap each other, like pieces of paper would on a regular desk.

Let's take a look at each window and learn its purpose.

The List window

The List window is *active* when BASIC is first started. When a window is active, it will receive anything you type on the keyboard. It's as if this particular piece of paper is on top of the heap, ready for your pen. You can tell it is active because its title bar contains horizontal lines; the title bars of the other windows don't have these lines, indicating that they are presently inactive.

The purpose of the List window is to hold our BASIC program. You can type text and numbers into it, erase what you've typed, and edit your entries in it. We won't teach you what a program is in any detail yet, but for now you can think of it as a collection of special words and instructions that tell BASIC what actions to perform. We type these words and instructions into the List window.

Now let's look at several specific features of the List window.

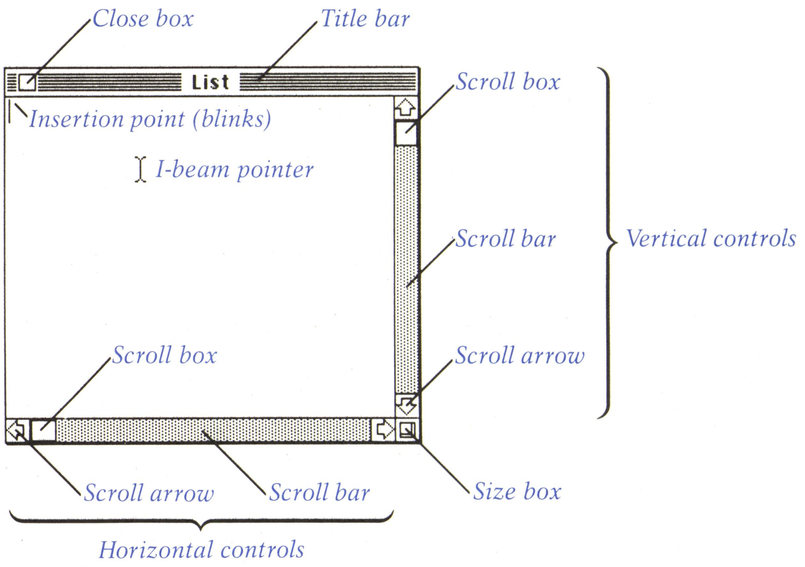
Scrolling the List window

Figure 1-9, on the following page, shows the details of the List window. When we move the mouse arrow-pointer into the List window, the arrow instantly changes shape and becomes an *I-beam pointer.* The I-beam pointer is used to move the *insertion point,* a vertical bar which marks the place where any typing or editing we do will appear, to different locations within the program.

The most notable features of the List window are the shaded *scroll bars* along the side and bottom edges. Imagine that a long piece of paper is located directly behind the List window and that the piece of paper contains BASIC program instructions. The List window lets us see just the text of the section of the program that the window is directly over. To see other sections of the program, we can use the scroll bars to move it under the

FIGURE 1-9

Elements of the List window


Close box
Title bar
List
Scroll box
Insertion point (blinks)
I-beam pointer
Scroll bar
Vertical controls
Scroll arrow
Scroll box
Scroll arrow
Scroll bar
Size box
Horizontal controls

window, like someone examining sections of a document with a magnifying glass. When we click and hold the arrow-pointer in the *scroll arrows* at either end of the vertical scroll bar, the text on the imaginary paper moves up or down one line at a time (or we can visualize the window moving up or down). The small white box located in the scroll bar is known as the *scroll box*. When we click on the scroll box and drag it, we move the program (our piece of paper) proportionally to its distance along the bar. For example, if the scroll box is dragged from the top of the vertical scroll bar to the middle, the window will move to about the middle of the program.

The vertical scroll bar is used for moving up and down the length of the program in the List window. The horizontal scroll bar is used for moving horizontally, when program lines extend beyond the right margin of the List window.

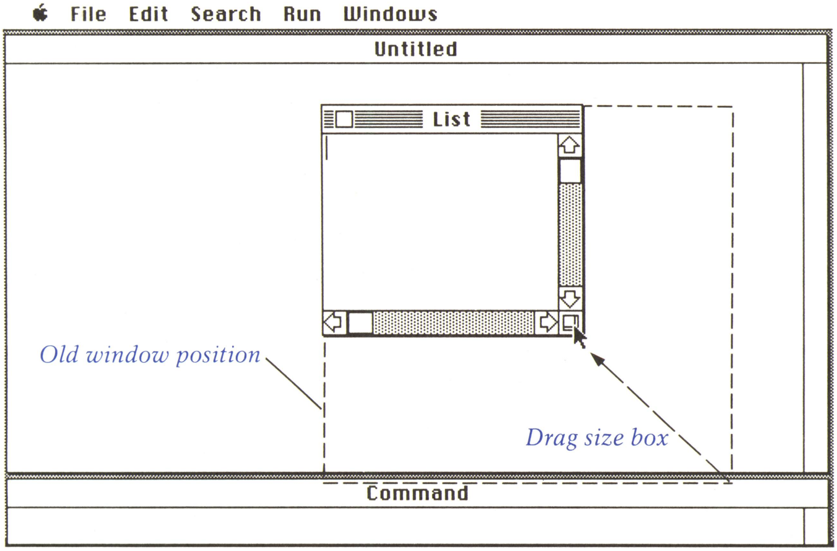
Resizing and moving the List window

The lower right corner of the List window contains the window's *size box*. This allows us to change the size of the horizontal and vertical dimensions of the window to suit our needs. To see how this works for the List window:

Drag:	the size box toward the top left corner of the List window
Release:	the mouse button to fix the new window size

FIGURE 1-10

Resizing the List window


File Edit Search Run Windows
Untitled
List
Old window position
Drag size box
Command

The List window changes size as we drag the size box. Figure 1-10 shows the resized List window.

Besides being able to change the size of the List window, we can also move it on the desktop by simply clicking and dragging its title bar.

Expanding the List window to full size

We can quickly resize the List window to fill the entire desktop by double-clicking in the List window's title bar, as shown in Figure 1-11.

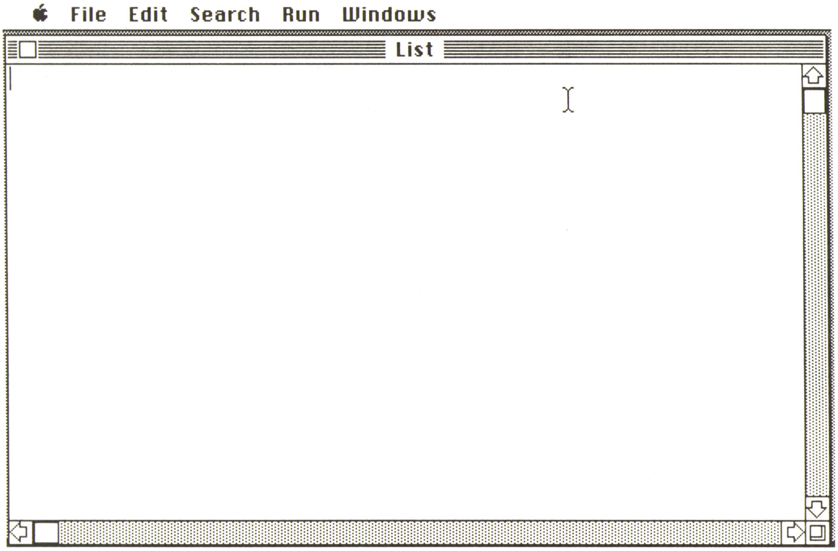

File Edit Search Run Windows
List

FIGURE 1-11

The expanded List window, after double-clicking in small window's title bar

And once we've expanded the List window to full size, we can return it to the size it was before expansion by double-clicking in its title bar again.

Two List windows

One extremely powerful feature of BASIC's windowing capacity is that there can be *two* List windows open on the desktop at once! The second List window is activated by the Window menu (we'll say more about that soon). Since it will contain the same program as the first List window, having two List windows does not mean we can run two programs at once. But it does mean we can look at two different parts of the same program at the same time and compare them. We can easily make text transfers between these two windows using the Cut, Copy, and Paste commands on the Edit menu, as we'll discover. This makes editing a program much easier and faster, since we don't have to keep scrolling from one end of a long program to the other.

The Command window

To *activate* a particular window and bring it to the front of the other windows, we click in it. Let's click in the Command window and watch what happens. The Command window becomes active and the List window inactive. The Command window is where we can type commands (BASIC statements) and test ideas. The Command window is different from the List window in that whatever is typed in the Command window is immediately executed by BASIC when we press the Return key, and then it is lost. On the other hand, information typed in the List window is saved.

As in the List window, the mouse arrow-pointer becomes an I-beam pointer when it's in the Command window, and we use it to move the insertion point to different places in our entries. And also like the List window, the Command window can be resized by dragging its size box, moved by dragging its title bar, and fully expanded and contracted by double-clicking in its title bar. There are no scroll bars because only one line can be typed in the Command window at a time, so there is no document to scroll through. We will learn more about using the Command window in the next chapter.

The Output window

Each window in BASIC performs a different service. The List window holds our program. The Command window lets us quickly try out simple ideas. The Output window displays the results of our programs when they are actually run, which can include text, numbers, and graphics of all types and shapes. Besides simply displaying text and graphics, we can use the

Output window to ask the user of our program questions. We can think of the Output window as a graphic link to the person using our creative and wonderful BASIC program. In later chapters, we will see that we can have more than one Output window open at once.

When we start up BASIC by double-clicking on its Macintosh-desktop icon, the Output window that appears is labeled *Untitled.* We will see later that when we *save* a program we've created, the Macintosh's operating system asks us to name it. It will then save the program on the disk with its own icon. Thereafter, whenever we open the program, the name we have chosen will appear in place of the word Untitled in the Output window's title bar.

The Output window can be resized, expanded and contracted, and moved using the same procedures described for the List and Command windows.

ARRANGING WINDOWS

We can position and size the various BASIC windows in many different ways. Figure 1-12 shows one way to arrange the windows. This example allows us to see both the output of a program when it's run, and part of the program itself. As we learn BASIC we will find that several factors affect where windows are positioned on the desktop.

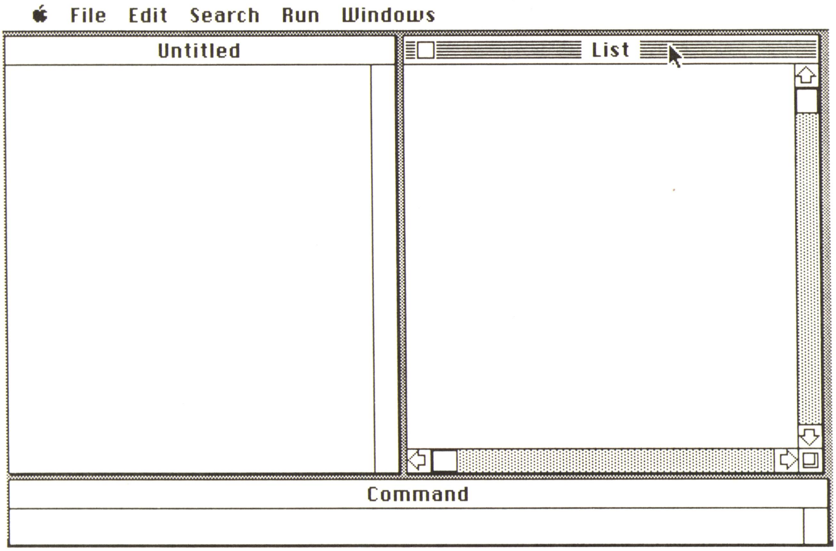


FIGURE 1-12

Windows rearranged on the BASIC desktop

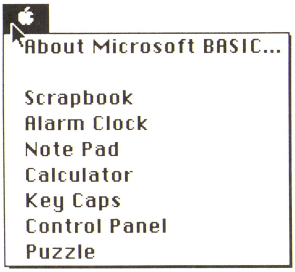
MICROSOFT BASIC MENUS

There are six menus on the Microsoft BASIC menu bar: Apple, File, Edit, Search, Run, and Windows. We cannot always use all of these menus at the same time. When a menu name is "dimmed"—its name is light gray—it means that the menu is not relevant to what we are doing at the moment. For example, earlier when we clicked on the Command window, you might have noticed that the Search menu title went dim. Use of the Search menu requires that the List window be active and when it is not active, the BASIC environment dims the Search menu title.

We will not give an exhaustive explanation of the menus now since you have no context in which to try them out. We will rather give you an overview of them. Complete details can be found in Chapter 3 of the Microsoft BASIC Interpreter manual.

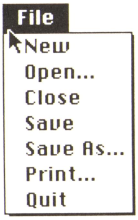
The Apple menu

The Apple menu is the first menu in the menu bar and is signified by a tiny Apple symbol. The menu contains what are called the Macintosh *desk accessories.* These are functions that Apple has provided that can be used from within Microsoft BASIC. Details about the desk accessories can be found in the Macintosh Owner's Manual.



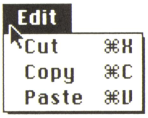
The File menu

The File menu is used to save and load our BASIC programs to and from *files* on the disk, and to allow us to quit BASIC and return to the Macintosh desktop. We can also print a *listing* of a program from this menu. We will learn exactly how to save, load, and print programs in Chapter 3.

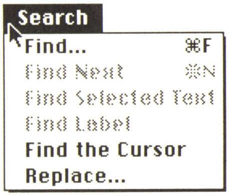


The Edit menu

The Edit menu is used when we are working in the List window on our program. The entries in the Edit menu allow us to move text around in the program, and to delete blocks of text from the program. We will cover this in more detail in Chapter 3.



Note that to the right of each command on the Edit menu is a funny symbol followed by a letter. The funny symbol represents the Command key located at the lower left of the keyboard, right next to the Option key. When a Command key is shown next to a menu command, it means we can press the given key combination instead of choosing the command with the mouse. When we are editing programs, we will often find it faster to use the Command-key combination instead of pulling down the menu and making a selection with the mouse.

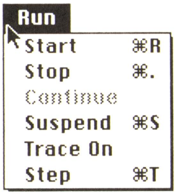


The Search menu

To see the Search menu, the List window must be active. The Search menu is used to automatically find and replace text in our programs. It comes in handy when we have a long program with a certain name or program statement that we need to change appearing in several places. By using the Search menu, we can have BASIC automatically find these names and change them for us. Note that several of the items in the Search menu are dimmed. They become active as soon as we have given the Search function something to search for. We will describe the Search menu when we have a program that requires it.

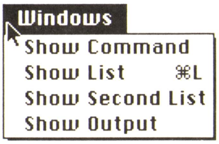
The Run menu

The Run menu is used to stop and start our BASIC programs, as we'll discover in Chapter 3. It is also used to *debug* our programs. Debugging means finding and fixing mistakes in our BASIC program logic.



The Windows menu

The Windows menu is extremely important because it lets us bring a window that has become "hidden" by other windows to the front and make it active. It also lets us open a window that has been closed by clicking in its *close box,* the small box at the left end of the window's title bar.



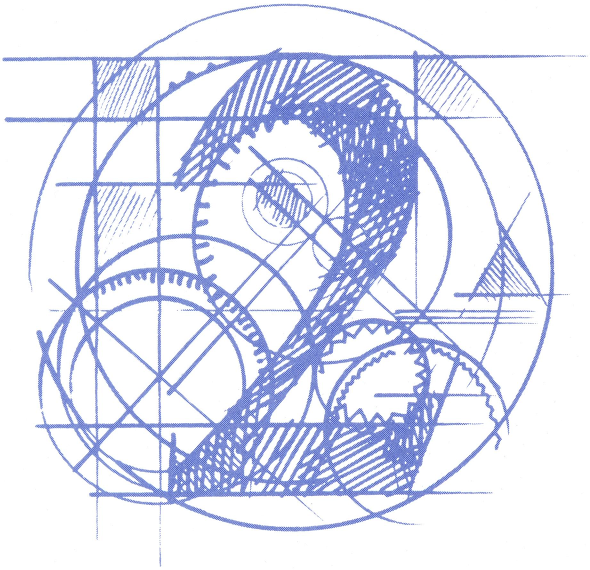
As an example, when BASIC is first started and the List window is on top of the Output window, clicking on the Output window automatically brings it to the forefront, obscuring the List window behind it. To make the List window come to the front, choose Show List from the Windows menu (or press its Command-key combination, Command-L) and presto, the List window reappears where it was.

We discovered earlier in this chapter that it is not always necessary to use the Windows menu to make a window active. If any part of an inactive window is visible, we can click on it and it will become active. Thus the Window menu is necessary only when we can't see the window we want to activate.

REFRESH-YOURSELF WINDOW

The end of the beginning is upon us. We have made a backup copy of our new BASIC disk, and learned about the two different versions of BASIC. We know our way around the BASIC desktop, and have had an overview of the windows and menus of BASIC.

In the next chapter we will begin our journey into BASIC's interesting collection of instructions.



Instant BASIC

In the last chapter we introduced you to the Microsoft BASIC desktop, which appears when we open BASIC. In this chapter we're going to practice using one of the windows on the desktop: the Command window. This is the easiest of the windows to use. You can type a single BASIC statement into the Command window, and the results will be immediately displayed in the

Output window. While learning how to work with the Command window, we'll also discover some simple but very important BASIC *statements.*

We're going to begin our journey into Microsoft BASIC by showing you how to use the PRINT statement to put words, numbers, and even the time and date in the Output window. We'll cover variations in the way PRINT is used, and we'll even show you how this statement can be used to turn your Macintosh into the world's most expensive calculator. And as a bonus, at the end of this chapter we'll discover that BASIC can do things unrelated to the Output window, by learning several BASIC statements that cause the Macintosh to make sounds.

Let's begin by turning on the Macintosh, inserting our BIN BASIC work disk, and double-clicking on the BASIC (b) icon in the Open window to start up BASIC (if the window isn't open, double-click on the disk icon to open it). The BASIC desktop soon appears with the List, Command, and Output windows that we discussed in the last chapter.

MAKING ENTRIES IN THE COMMAND WINDOW

The first thing we want to do is make the Command window active. Remember how?

Click:	on the Command window

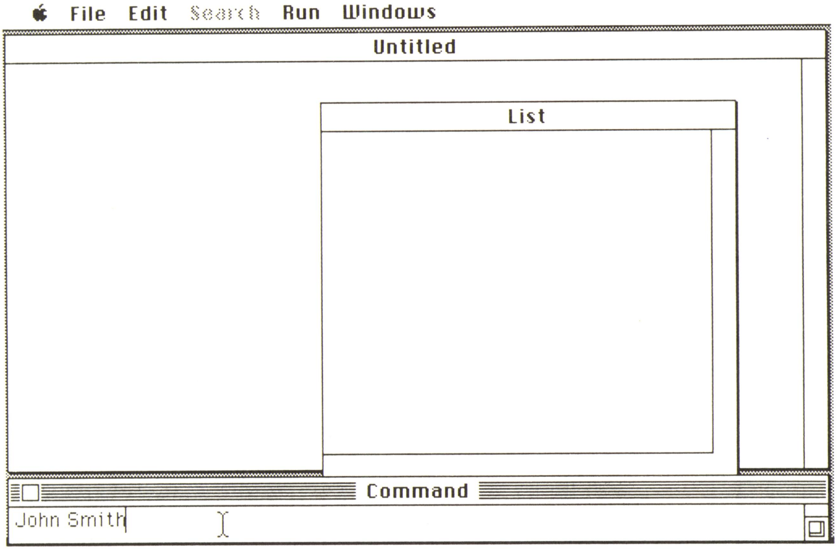
A band of closely spaced horizontal lines appears in its title bar, and the blinking insertion point appears on the left side of the Command window. Whatever we type will appear at the position marked by this insertion point, and the insertion point will march to the right as the letters we type appear in the window.

Now we're ready to type something into the Command window. How about a name, like John Smith.

Type:	John Smith

Simply type the letters as you would on a typewriter. The result should look like Figure 2-1.

When we're done, nothing happens. Why not? Because BASIC doesn't know we're done typing. The insertion point is still blinking just


File
Edit
Search
Run
Windows
Untitled
List
Command

John Smith

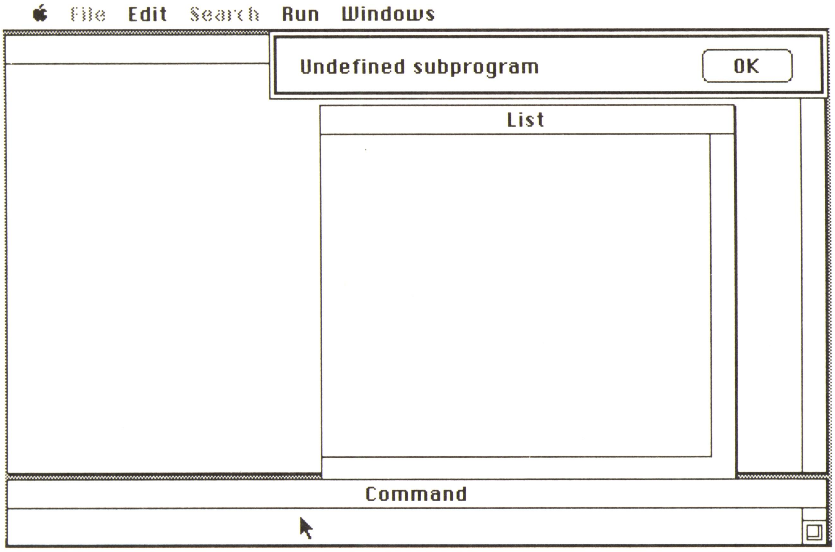
FIGURE 2-1

A name typed in the Command window

past the last letter we typed, showing that it's waiting for us to continue typing. How do we tell BASIC we're done? By pressing the Return key.

Press: the Return key

Oh, dear—what happened? The computer beeped at us, and the *error box* shown in Figure 2-2 appeared with the mysterious *error message* "Undefined subprogram."


File
Edit
Search
Run
Windows
OK
List
Command

Undefined subprogram

FIGURE 2-2

The undefined subprogram error message

What in the world does this error message mean? For the moment, all you need to know is that it is one of the ways BASIC has of telling us it doesn't recognize what we typed.

RESERVED WORDS, STATEMENTS, AND FUNCTIONS

The only words that BASIC recognizes are certain special words that often look like abbreviated English, like PRINT, INPUT, and CIRCLE. Sometimes the words are abbreviations that don't much resemble English, like CLS and ABS. Sometimes they look like English except for the addition of a special symbol, like RIGHT$.

The special words that BASIC understands are called *reserved words,* meaning that they are reserved exclusively for use by BASIC (although we can use them within quotation marks, as we'll discover). The reserved words are combined in various ways to create *statements* and *functions* that instruct BASIC to do something. A statement instructs BASIC to perform an action, such as printing a sentence on the screen. A function tells BASIC to acquire, or "return," some piece of information, such as finding the square root of a number. We're going to explain these reserved words one at a time as we go along, so that by the end of this book you'll know all the major ones. (A complete list of these reserved words can be found in Appendix C of the Microsoft BASIC Interpreter manual that came with your program disk.)

But to get back to our error message: The name we entered was not on the list of reserved words that BASIC knows, so it put an error message on the screen. To get rid of the error box:

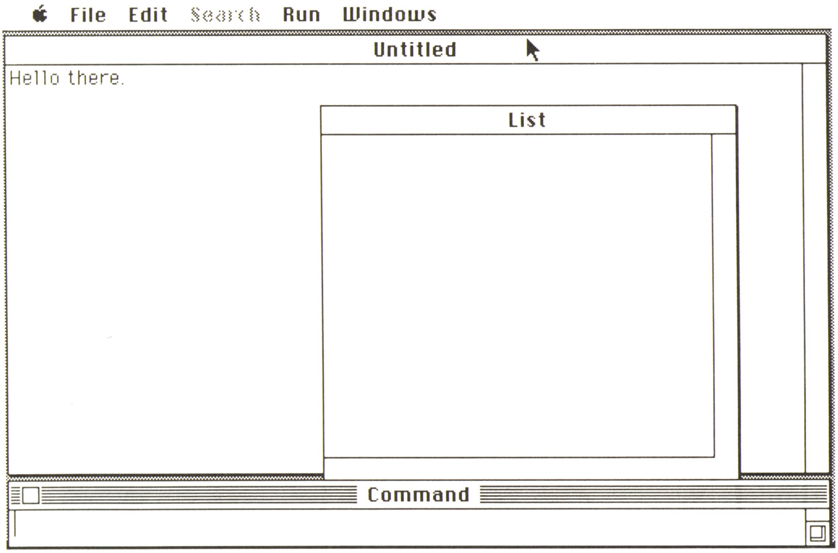
Click:	the OK button in the error box

The error box goes away and the Command window is cleared of whatever we typed. Now we're back where we started, and we're ready to see if we can type something that BASIC *does* understand.

The first special BASIC reserved word that we're going to look at and try out is PRINT.

THE PRINT STATEMENT

The Command window should still be active; if it isn't, click on it to make it active. Now we're going to type a legal BASIC statement into the Command window, and this time it will be one that BASIC does understand.



```
Hello there.
```

FIGURE 2-3

The result of the BASIC statement: print "Hello there."

Type: print "Hello there."

Type this exactly as shown: first *print* and then the phrase surrounded by quotes. If you make a mistake, you can use the Backspace key to erase as many letters as necessary, and then type it again. When you're done, use the Return key to tell BASIC that you're ready for it to go to work.

Press: [Return]

The screen now looks like Figure 2-3: The phrase "Hello there." appears at the top of the Output window, and the statement we typed in the Command window vanishes so we can type something else.

For the first time, BASIC has carried out our instructions! We've actually written a statement in the BASIC programming language. That wasn't so hard, was it? To prove that it wasn't a fluke, type this statement in the Command window:

Type: print "Welcome to Microsoft BASIC."
Press: [Return]

The new phrase is printed directly below "Hello there." in the Output window. That was even easier than the first phrase, wasn't it?

Type in some phrases of your own. Remember, always start with the BASIC statement PRINT, then quote marks, then the phrase you want to print, and finally close quotes. You can print anything you like in the Output window: letters, numbers, even some special characters.

When you PRINT in the Output window, the List window remains on top of the Output window. If what you print is too long, it may disappear behind the List window. If this happens, you can resize and move the List window to get it out of the way, or simply click on the Output window to send the List window "behind" it. However, what was printed "behind" the List window will not reappear until you repeat the print statement.

A PRINT shortcut

Sometimes you will be faced with having to type lots of PRINT statements. Typing PRINT over and over can be tiring. Microsoft has provided a single-character substitute for the PRINT command-word: the question mark. When BASIC sees a question mark *outside* of quotation marks, it assumes that we want it to perform a PRINT command.

UPPER- AND LOWERCASE

When you type a BASIC statement like PRINT, it doesn't matter at all to BASIC whether you type it in upper- or lowercase, or any combination thereof. PRINT, Print, print, and prINt are all the same to BASIC. It's easier to type in lowercase since we don't need to hold down the Shift key, so that's how we show it when we tell you what to type. However, when we mention a statement like PRINT in a sentence such as this, we will show it in all uppercase letters so it will stand out from the "normal" English words. The moral is that when you see a BASIC statement in uppercase in the text, it doesn't mean you have to type it in uppercase.

BASIC STRINGS

The phrase inside the quote marks following PRINT in each of the examples we have used so far is called a *string* because it's made up of a series of letters and other characters such as punctuation marks—kind of like a "string of pearls." A string can be shorter or longer than a sentence, and can include letters, punctuation, and the numerals from 0 through 9, taken in a specific order. A string can be practically any length, as we shall see.

CLEARING THE SCREEN: CLS

Before we go on to explore other uses of PRINT, it would be nice to get rid of all the phrases we've been printing in the Output window. Depending on how enthusiastic you were in exercising the PRINT statement, the screen may be practically choked with clever phrases. Is there any way to erase something that's already been printed?

As it turns out, there's a BASIC command that erases the entire Output window. The command is CLS, which stands for *clear screen.* (It doesn't really clear the *entire* screen—but the command's name is left over from the days when computers didn't have separate Output windows, and then the whole screen *was* cleared.)

To clear the Output window now, make sure the Command window is active, then:

Type: cls
Press: [Return]

That's all there is to it. Any letters, numbers, or words in the Output window vanish. We have completely cleaned off our Output window, and we're ready to print something fresh and new. (*Note:* To save space, we're sometimes going to include the instruction to press the Return key on the same line as the instruction for what you should type.)

PRINTING NUMBERS AND ARITHMETIC IN BASIC

Make sure that the Command window is active. Now:

Type: print 7

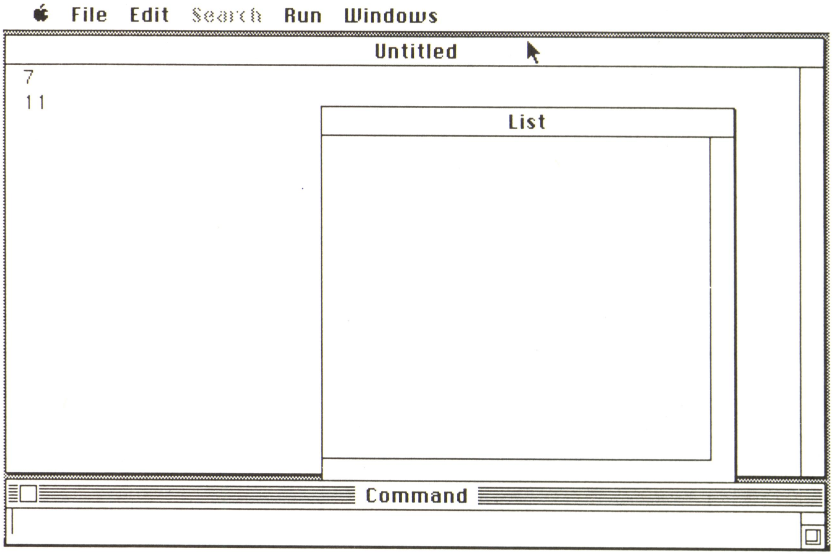
Notice that we didn't put quotes around the 7, and just typed it as a plain old vanilla number.

Press: [Return]

What happens? Nothing startling: A 7 is printed in the Output window. That's just what you expected, isn't it? But what about this:

Type: print 7 + 4[Return]

Now something interesting appears. Instead of printing 7 + 4 in the Output window as you might have expected it to do, the computer actually did the



```
7
11
```

FIGURE 2-4

Arithmetic in the Output window

addition: It added 7 to 4 and printed the sum, 11. The resulting screen looks like Figure 2-4.

What's going on here? It turns out that in Microsoft BASIC, the arithmetic function 7+4 is an example of an *expression.* An expression can be a number, but it can also be a combination of numbers and operators like plus or minus signs. Now type another expression in the Command window:

Type: print 17−7[Return]

The number 10 appears in the Output window, since we've subtracted 7 from 17. How about multiplication?

Type: print 3 * 4[Return]

This gives us 12, which is 3 times 4. As you can see, the symbol for multiplication isn't the one you're probably familiar with: the letter *x*. Because *x* might be confused with a variable of the same name (we'll tell you about variables in future chapters), the asterisk (*) is used to indicate multiplication. To round out the arithmetic operators, let's try division:

Type: print 33/3[Return]

The number 11 appears in the Output window, since it's the result of 33 divided by 3. For division, the operator BASIC understands is the slash (/) and not the familiar ÷ sign.

You can print out an expression that's as complicated as you like. For instance:

Type: print 2 * 4 + 5 * 3

You should get 23, since 2 times 4 is 8, 5 times 3 is 15, and 8 plus 15 is 23. Why didn't BASIC add the 4 and 5 together before multiplying? The answer is that there are certain rules about the order in which BASIC performs arithmetic calculations. Thus, when BASIC confronts a series of arithmetic operations, it first *multiplies* or *divides* the numbers on either side of any multiplication or division signs, then performs any leftover *addition* and *subtraction* operations. If all of the operations are of equal precedence, such as a series of multiplication and division operations, BASIC calculates them in order from left to right.

But we don't have to accept the precedence of operations set by BASIC. If we enclose a sequence of operations within parentheses, BASIC first calculates the results of those operations, then uses the results to finish calculating the operations outside of any parentheses. If we:

Type: print (2 + 2) * (6 – 4)[Return]

BASIC calculates and prints the result as 8. Now if we rearrange the parentheses in the numerical expression:

Type: print 2 + (2 * 6 – 4)[Return]

we get a quite different result: 10. So by judicious use of parentheses, we can perform any calculation in any order that we choose.

There's more to be said about numerical expressions, but we've shown you enough to give you a flavor for the ways you can use PRINT with numbers. We'll return to some of the more sophisticated uses of numerical expressions in later chapters.

The invisible plus sign

You may have noticed that the numbers we've printed don't appear close against the left edge of the window the way the strings did. Why is this? To learn the answer:

Type: print 7 – 10[Return]

This time we find the answer printed close against the window's left edge. That is, the *minus sign* in −3 is close against the edge. The number itself is one character away.

It turns out that BASIC always leaves space for a sign. If the sign is negative, it prints a dash in this space. If it's positive, it doesn't print anything, but it leaves the space just in case. This is a help when you're trying to line numbers up in columns, as we'll show you in the next section.

USING COMMAS IN PRINT STATEMENTS

We're not limited to printing only one string or one number or one expression at a time in the Output window. In fact, we can print a number of these elements on the same line using a single PRINT statement. The trick is to use various punctuation marks as signals to the PRINT statement to combine things in different ways. For instance, using *commas* causes the output to be arranged in columns. Let's see what this means.

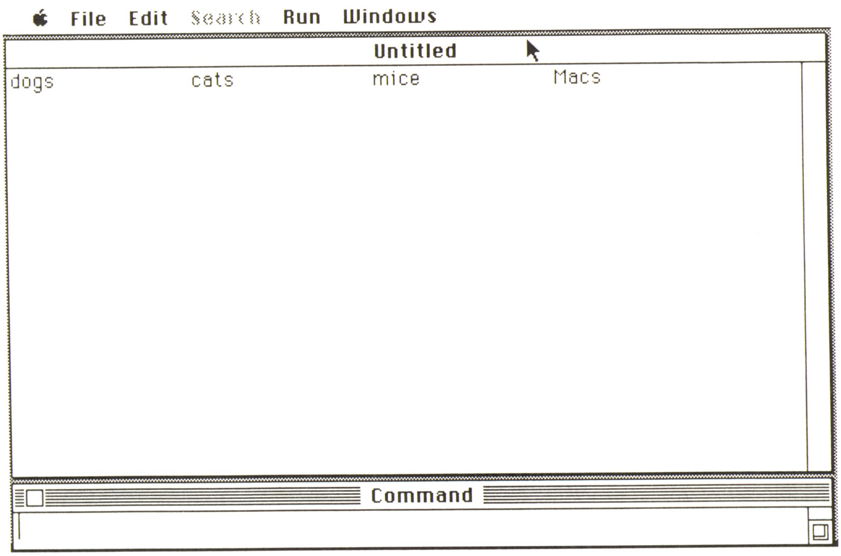
First, let's make sure we've got room to work on the Output window by getting rid of the List window. This is easy: Just click on the close box in the upper left corner of the List window to make it vanish. Then, clear the Output window by typing CLS in the Command window. Now we're ready to see what commas can do for PRINT.

Type: print “dogs”, “cats”, “mice”, “Macs”

Take care when you type this, because there are some pitfalls for the unwary. Notice that each word is surrounded by quote marks, and that the commas are *outside* of the quote marks, not inside as you would expect. Also notice that there are no spaces inside the quotes. To see what this statement does:

Press: [Return]

The result is that the four words are printed in regularly spaced columns across the Output window, as shown in Figure 2-5. The positions of these columns have been chosen for us by BASIC: They're 14 characters apart. The effect is very similar to using the tab settings with a typewriter or a word-processing program like MacWrite or Microsoft Word.



```
dogs              cats              mice              Macs
```

FIGURE 2-5

The result of using commas in a PRINT statement

Without erasing the Output window, type these numbers in the Command window (remember, we don't need to type any quote marks around numbers):

Type: print 1, 2, 3, 4[Return]

The numbers line up under the previous words we printed: The commas work the same way when printing numbers as they do with strings. (The numbers actually line up under the second letter of each word, because space is reserved for the invisible plus sign that precedes the number.)

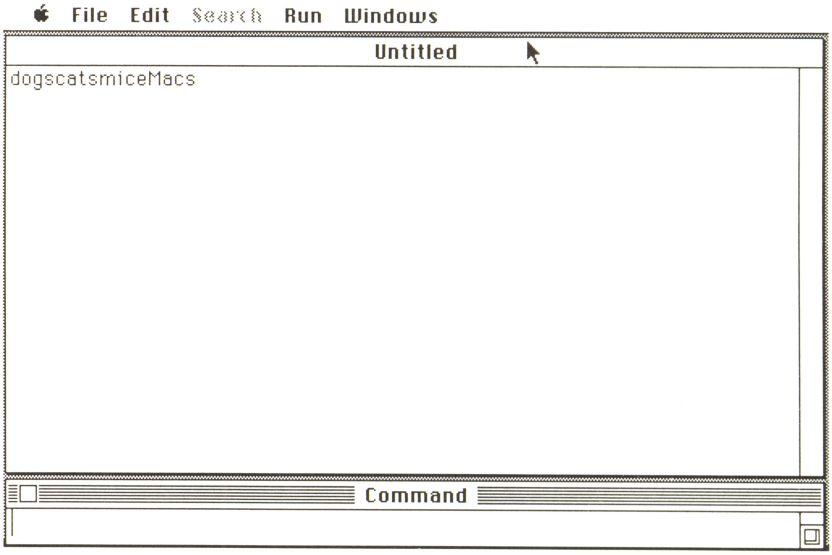
USING THE WIDTH COMMAND

We can change the width of the columns if we want, by using the WIDTH command in a statement. Try this:

Type: width, 8[Return]
Type: print 1, 2, 3, 4[Return]

This time the numbers are printed closer together because we've made the columns 8 characters wide instead of 14. To get the width of columns back to 14 characters, type *WIDTH, 14.*

Later on we'll show you situations where the WIDTH feature can come in handy, and we'll describe it more fully then. For the moment, let's look at the second major way to control output in PRINT statements.



```
dogscatsmiceMacs
```

FIGURE 2-6

The result of using semicolons in a PRINT statement

USING SEMICOLONS IN PRINT STATEMENTS

Suppose we don't want numbers or words to come out in columns, but instead want them to appear right next to each other on the screen. How do we communicate this desire to PRINT? By using the semicolon.

To see the effect of using semicolons in a PRINT statement, clear the Output window with the CLS command, then retype the cats and dogs line, but with the commas replaced with semicolons:

Type: print "dogs"; "cats"; "mice"; "Macs"[Return]

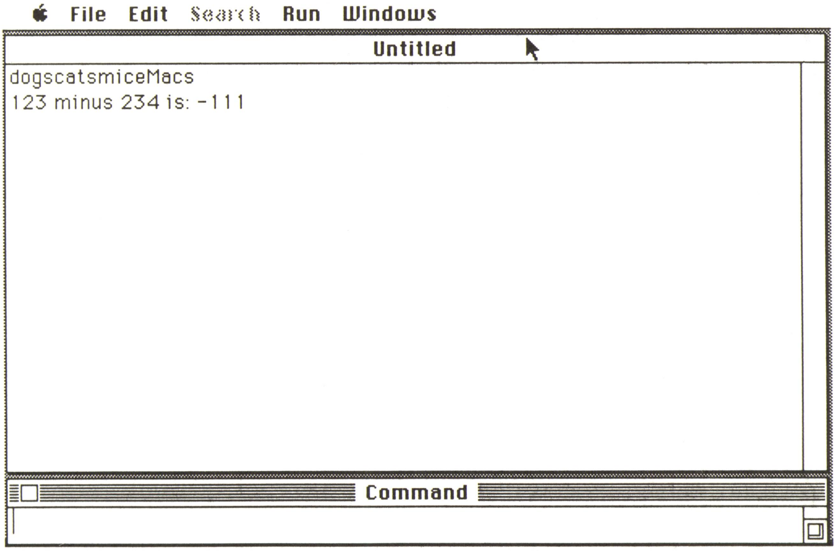
The resulting Output window looks like Figure 2-6.

Combining strings and expressions in PRINT statements

That's fine, but now all the words are run together. What good is that? The moral is that you need to think about *spaces* when you use the semicolon option with PRINT. Let's try an example where using the semicolon makes more sense:

Type: print "123 minus 234 is: "; 123 – 234[Return]

Here we've printed a string (the phrase in quotes), followed by a semicolon, followed by an arithmetic expression. The string happens to state the same problem that BASIC solves when it looks at the expression: How much do you get when you subtract 234 from 123? Since the result is a negative number, a minus sign is printed before it, as shown in Figure 2-7.



```
dogscatsmiceMacs
123 minus 234 is: -111
```

FIGURE 2-7

A string and an expression connected with a semicolon

Note that it's important to include the space after the colon following *is*; otherwise the minus sign butts right up against the colon and is hard to read. However, if the expression *precedes* the print string, you don't need to include the space because one is always printed following a number. Let's reverse the order of the expression and string we just printed:

Type: print 123 – 234; "is 123 minus 234."[Return]

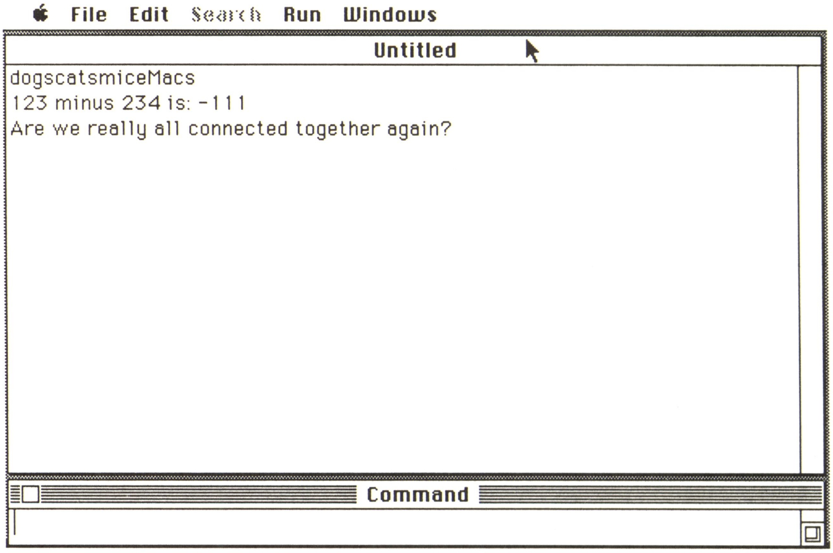
Even though we didn't include a space between the first quote mark and *is*, one space was inserted between the number –111 and *is* in the resulting printed line.

Using semicolons at the end of PRINT statements

A special thing happens when the semicolon is the last symbol in a PRINT statement. Let's type this:

Type: print "Are we really all connected ";[Return]
Type: print "together again?"[Return]

When a semicolon is put at the end of a PRINT statement, BASIC "remembers" where the last letter or symbol was printed. When the next PRINT



```
dogscatsmiceMacs
123 minus 234 is: -111
Are we really all connected together again?
```

FIGURE 2-8

The result of using a semicolon between separate PRINT statements

statement occurs, its text starts printing where the last left off. The result, shown in Figure 2-8, is a "normal," unbroken sentence.

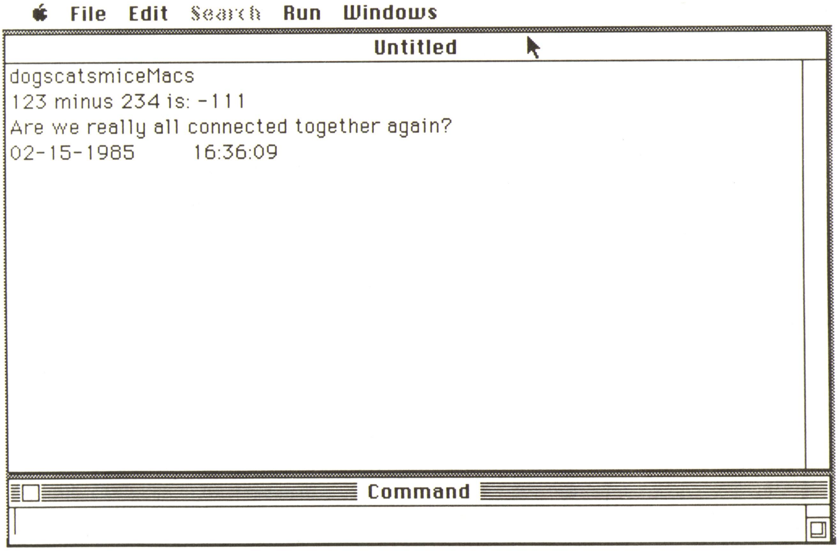
THE DATE$ AND TIME$ FUNCTIONS

Before we leave the PRINT statement, we're going to show you two more of the many other things you can print with it: the date and the time. DATE$ and TIME$ are called *functions* because they produce certain results for us, and the dollar sign indicates that they deal with strings. The DATE$ and TIME$ functions can be useful features in BASIC programs that generate reports, since the current date and time can be printed automatically on the report by the program.

In order for the DATE$ and TIME$ functions to produce the correct results, you must have already set the Macintosh's battery-powered internal clock to the correct date and time on the Control Panel desk accessory under the Apple menu. This process is described in detail in the manual that came with the Macintosh. Having taken care of this detail, make sure the Command window is active and:

Type: print date$, time$[Return]

You should see the current date and time displayed in the Output window of your Mac, similar to Figure 2-9.



```
dogscatsmiceMacs
123 minus 234 is: -111
Are we really all connected together again?
02-15-1985      16:36:09
```

FIGURE 2-9

The result of using DATE$ and TIME$ in a PRINT statement

As you can see, the month, day, and year of the date are separated by dashes, and the time is separated by colons. Also, the time is printed with 24-hour notation, which allows BASIC to dispense with the a.m. and p.m. specifiers.

THE MACINTOSH SPEAKS: BEEP AND SOUND

Before we go on to the next chapter, we want to broaden your perspective by showing you that BASIC statements can affect things other than the Output window. We'll do this using two statements that cause the Mac to make sounds.

The BEEP statement

First, let's find out what the BEEP statement does. Make sure the Command window is active:

Click: on the Command window
Type: beep[Return]

We can't show a figure of the result as we usually do, since nothing happens in the Output window. However, your computer should make a short, pleasant tone. Later, when we learn to write programs, we'll find that this tone is just what we need when we want to get the attention of the person using our program.

The SOUND statement

You can't change the beep in the BEEP statement. It always has the same pitch and the same duration. The SOUND statement gives you more flexibility: You can make it a high or low pitch, and you can make it very short or very long.

The SOUND statement is followed by two numbers. These numbers represent the desired *pitch* and *duration* of the sound. In other words, the first number in SOUND specifies the frequency of the sound, and the second number specifies the length of time the tone will last. Try this in the Command window:

```
Type:     sound 80, 40[Return]
```

You should get a low tone which lasts considerably longer than the tone generated by the BEEP statement. For a higher pitch tone, try this:

```
Type:     sound 800, 40[Return]
```

This is a higher-pitched tone, because the 80 has been changed to 800, so the pitch went from 80 cycles to 800 cycles, but the tone lasts the same length of time because the duration number is the same. To make a shorter sound, try this:

```
Type:     sound 800, 5[Return]
```

The sound lasts only ⅛ the time it did before, because 5 is ⅛ of 40.

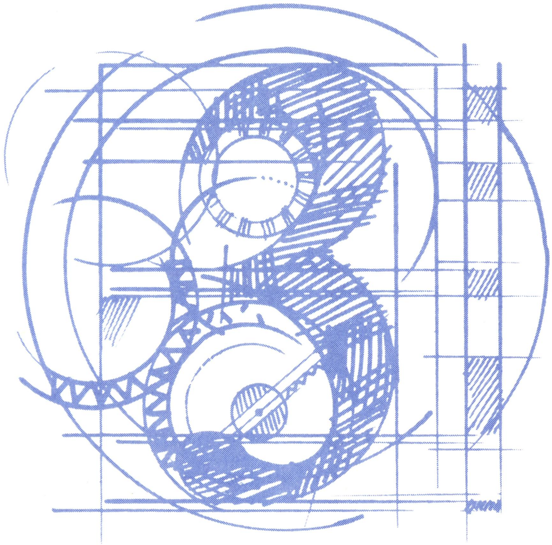
There are many other things you can do with the SOUND statement. It can be used to generate music and sound effects for games. The Macintosh has the ability to make four different sounds simultaneously, and each sound can have its own *voice*. We'll show you how to do this in a later chapter.

REFRESH-YOURSELF WINDOW

So far we've shown you all sorts of interesting things you can do using the Command window, including printing from it to the Output window, and making sounds.

BASIC statements typed into the Command window have the advantage that as soon as we press Return, the statement is executed and we can see (or hear) the result. We can do some simple things this way: For example, we can make our computer act like a one-line typewriter or a pocket calculator, or cause it to make a single musical tone. Later we'll find that this can be useful to test out a particular statement when we're actually writing a program.

However, the full power of our computer can only be realized if we can execute more than one statement line at a time. We need to know how to put several different BASIC statements together into a list called a *program.* This is what we can do in the List window, and is the subject of the next chapter.



Programs: Creating, Editing, and Saving

Now that we know about BASIC statements—how they work, and how to type them into the Command window to test them out—we're ready to learn how to combine a group of these statements into a program. *A program is what we get when we combine several*

BASIC statements in a certain order and have them execute in a series, one after the other.

We will create a very simple program and then try out some of the things we can do with it, such as running it, editing it, and saving it on disk. We need to know these things because they are tasks that are common to all programming.

PROGRAMMING IN THE LIST WINDOW

In the last chapter we typed BASIC commands in the Command window. We found that we can only type one BASIC statement at a time in this window. As soon as we press the Return key, the statement is executed and the entry disappears, leaving the Command window empty.

The List window is quite different. Whatever we type in this window stays there, even after we press Return at the end of a line. If you are at all familiar with the Microsoft Word or MacWrite word processors, or the Note Pad desk accessory, then you will find using the List window easy. If you are not familiar with these text editors, don't worry, because we will provide all you need to know.

By now you know that we have to press the Return key after every program line, so from now on, we're going to dispense with putting it in the instructions. Be careful not to press Return at the end of a line until you're sure it's accurate. If you make a mistake in a line, backspace over the mistake and type the line again, just as you did for mistakes in the Command window. If you find a mistake after you've finished the line and pressed Return, you'd better start the program over again from the beginning. How do you start over?

Starting over with the New command

Although the Output window can be cleared by using the CLS command, there is no similar shortcut for clearing the List window. But if we find ourselves with a mess in the List window that we want to get rid of completely so we can start over again, we can choose New from the File menu.

When we choose New, the disk turns, and the menu names flash briefly as Microsoft BASIC clears all information from the List, Command, and Output windows. If the List window was behind another window, it comes to the front and becomes active. Any statements that were in the List window are erased and we can start a new program.

Later in this chapter we will find out how to *edit* programs, so that we can avoid the inconvenience of having to start over every time we create a mess.

CREATING OUR FIRST PROGRAM

We're going to write a program that prints a fancy "title" on the Output window. You might find this sort of thing useful for printing titles in the Output window at the start of a program, or for printing titles on hard-copy printouts on your printer. (We'll learn how to use a printer with BASIC later in the book.) To give a little variety to our program, we'll have it beep both before and after writing the title on the screen.

Let's start typing the program. Don't forget that what we type is going to go in the List window, not the Command window. If you've just started BASIC, the List window is active, and the insertion point is waiting in the upper left corner for you to start typing. If it isn't, click on the List window to make it active, or choose Show List from the Windows menu. Then type the first line of our program:

```
Type:      beep
```

After we press Return, this entire BASIC statement remains in the List window, and the insertion point drops down to the beginning of the next line, where it waits for us to type something else. More surprisingly, the word BEEP is automatically capitalized and made boldface. We will find that all reserved words are redisplayed in bold capital letters in the List window. This helps us see more clearly what the program is doing. Now for the next program line:

```
Type:      print "*********************"
```

After we press Return, the 21 asterisks in this line appear immediately below the BEEP line in the List window. Let's continue on with the next three lines:

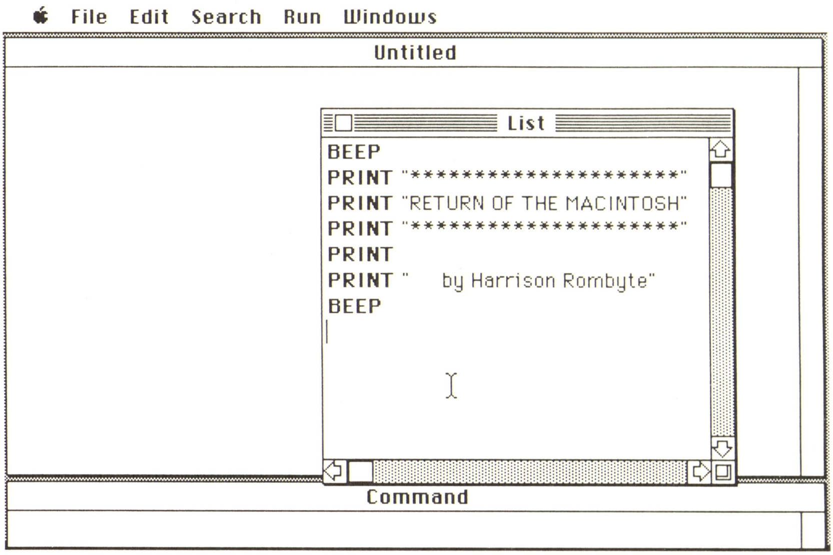
```
Type:      print "RETURN OF THE MACINTOSH"
Type:      print "*********************"
Type:      print
```

This last PRINT statement with nothing after it causes a line to be skipped in the Output window, as we'll see when we run the program.

```
Type:      print "     by Harrison Rombyte"
```

There are five spaces after the first quote marks in this line. And here comes the final line:

```
Type:      beep
```



```
BEEP
PRINT "*********************"
PRINT "RETURN OF THE MACINTOSH"
PRINT "*********************"
PRINT
PRINT "    by Harrison Rombyte"
BEEP
```

FIGURE 3-1

Our first program in the List window

When you've typed in these seven statements, your screen should look as shown in Figure 3-1. If your program looks different, choose New from the File menu and start over.

Remember that these statements are not executed when we type them in, as they were in the Command window. The list of statements in the List window is often called a program *listing.* The statements don't actually do anything until we tell them to, which is what we're going to learn about next.

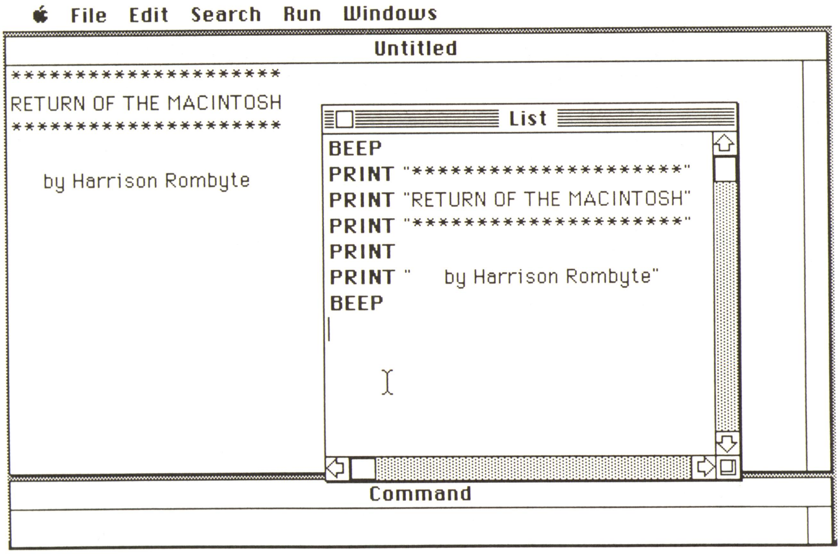
RUNNING OUR PROGRAM

Now that our program is completed and waiting patiently in the List window, we can run, or *execute,* it. This means that BASIC will go through the statements one by one, and do what each statement tells it to do.

Let's run the program and see what happens. To run the program, we use the Start command, which is one of the options in the Run menu.

Choose: Start from the Run menu

Wow! As soon as we release the mouse button on Start, the Mac beeps, the disk spins, the entire title is printed in the Output window as



```
**********************
RETURN OF THE MACINTOSH
**********************

    by Harrison Rombyte
```

```
BEEP
PRINT "**********************"
PRINT "RETURN OF THE MACINTOSH"
PRINT "**********************"
PRINT
PRINT "    by Harrison Rombyte"
BEEP
```

FIGURE 3-2

The output of our first program

shown in Figure 3-2, and we hear the second beep. There's not much time between the beeps, since the Mac executes the program so fast. The List window actually disappears while the program is running, and then reappears when the program is finished.

Want to run the program again? No problem—just go through the same sequence of operations: Pull down the the Run menu and choose Start. The Output window will clear, we'll hear the beeps, and the same output will be written again. We can do this as often as we like, without typing in any program statements! This is one of the nice things about a program: Once we've typed it in, the computer will faithfully execute it whenever we want, with no additional typing on our part.

EDITING OUR PROGRAM

A few paragraphs ago we told you that if you found a mistake in a program line after you pressed Return, your best bet was to start over and type the entire program in again. But this isn't the best way to correct an error in a program; if it were, programming would be much more difficult than it is, especially writing longer programs. Fortunately, Microsoft BASIC has some very sophisticated editing features, which we're going to learn about in this section.

Actually, Microsoft BASIC's editing features are very similar to those found in MacWrite, Microsoft Word, and the Note Pad desk accessory. If you already know how to use any of these, then you know most of what we're going to tell you and you can probably skim through this section rather quickly.

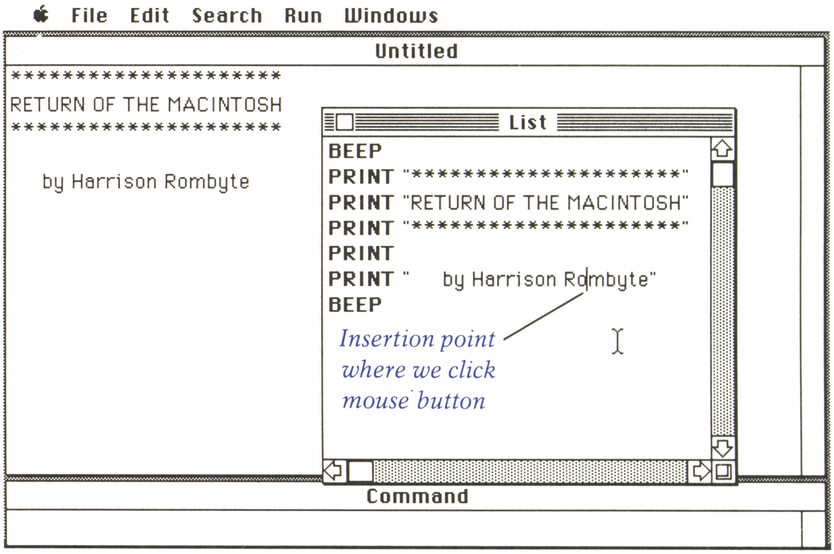
Positioning the insertion point

We can actually move the insertion point to any position in any previously typed line in the List window. This permits us to add or delete text anywhere we want. Let's see how to do it.

As we mentioned, the mouse arrow-pointer changes to an I-beam whenever we move it into the List window while the List window is active. We can use the I-beam pointer to set the insertion point wherever we want in the program listing. To move the insertion point to a particular place in the List window, simply move the I-beam pointer to the appropriate place in the program and click the mouse button. The insertion point starts blinking at that spot. Suppose, for instance, that we find we have misspelled the name Rombyte, and want to change it to Rambyte.

Click:	in the List window to make it active
Move:	the I-beam to just after the "o" in Rombyte
Click:	the mouse button

The insertion point now appears between the *o* and the *m* in Rombyte, as shown in Figure 3-3.



```
*********************
RETURN OF THE MACINTOSH
*********************

    by Harrison Rombyte
```

```
BEEP
PRINT "*********************"
PRINT "RETURN OF THE MACINTOSH"
PRINT "*********************"
PRINT
PRINT "    by Harrison Ro|mbyte"
BEEP
```

Insertion point where we click mouse button

FIGURE 3-3

Positioning the insertion point

Now we can use the Backspace key to erase the offending *o,* and then type the letter *a.* The name is changed just as we wanted.

Positioning the insertion point with the I-beam pointer, using the Backspace key to delete letters, and typing in what we want to add handles most common editing problems.

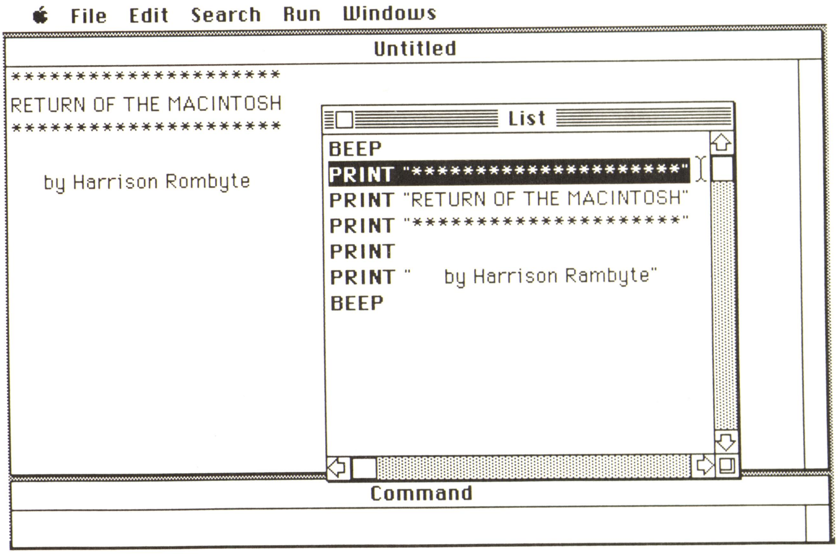
We can delete as many characters to the left of the insertion point as we wish, and we can even delete an entire line by positioning the insertion point at the end of the line and holding down the Backspace key until the line is gone. If we keep the Backspace key held down, when the insertion point reaches the left edge of the List window, it jumps to the end of the line above and continues erasing that line.

Selecting text

Deleting an entire program line by holding down the Backspace key can be rather slow. There is a faster technique that's also useful for other editing operations, such as cutting and pasting, which we'll talk about soon.

This faster deletion technique involves *selecting* text. To select a section of text in a program, we hold down the mouse button and *drag* the I-beam pointer over the text. Whatever text the I-beam pointer passes over turns black, or is *selected* in Macintosh terminology. Whenever we need to select just a single word, there's a shortcut we can use: Simply place the I-beam pointer anywhere within the word and quickly double-click the mouse button to make the word black and selected.

Suppose we decide that the title we're printing with our program would look better if we got rid of the top line of asterisks. To achieve this effect we need to delete the entire second line of our program. To delete this line, we'll select it by positioning the insertion point at the beginning of the line, hold down the mouse button, then move the mouse to the right to drag the pointer to the end of the line. As we move the pointer, the part of the line that the mouse passes over will change to white letters on a black background. This black area will be what we've selected. When we



```
*********************
RETURN OF THE MACINTOSH
*********************

    by Harrison Rombyte
```

```
BEEP
PRINT "*********************"
PRINT "RETURN OF THE MACINTOSH"
PRINT "*********************"
PRINT
PRINT "    by Harrison Rambyte"
BEEP
```

FIGURE 3-4

Selecting a line of text

get to the end of the line, we'll release the button. The entire line should be black, or selected, as shown in Figure 3-4.

Drag: the pointer across the first print statement:
PRINT “*********************”

If you make a mistake, and want to deselect what you've selected, simply click the mouse button while the I-beam pointer is in the List window. The white-on-black selected text changes to normal black-on-white, and you can start over.

Deleting selected text

Now that we've selected the first line, how do we delete it? Simple: by pressing the Backspace key. Just as it deletes one character, Backspace also deletes anything that's been selected.

Press: [Backspace]

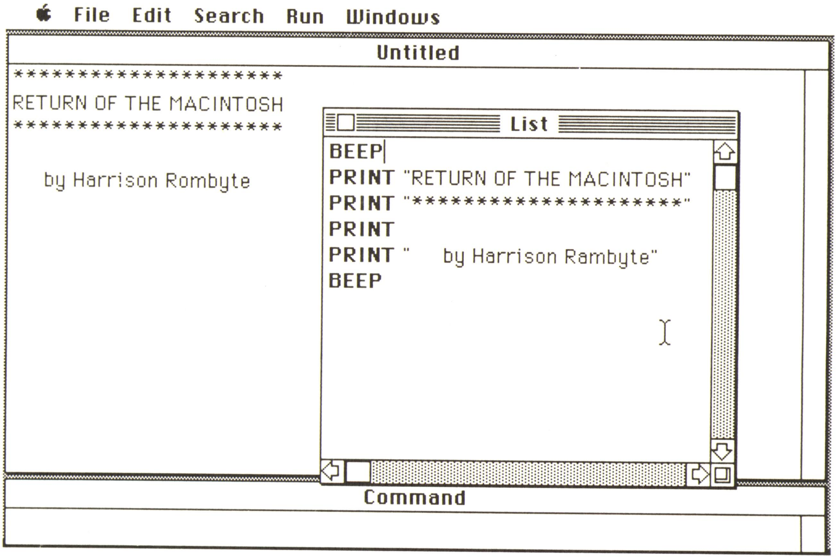
The line is eliminated. But in its place is a rather unattractive gap between the first and second lines of the program. To close the gap:

Press: [Backspace]

Now our program is six lines long, as shown in Figure 3-5.

FIGURE 3-5

The program with one line deleted



```
*********************
RETURN OF THE MACINTOSH
*********************

  by Harrison Rombyte
```

```
BEEP
PRINT "RETURN OF THE MACINTOSH"
PRINT "*********************"
PRINT
PRINT "    by Harrison Rambyte"
BEEP
```

Replacing selected text

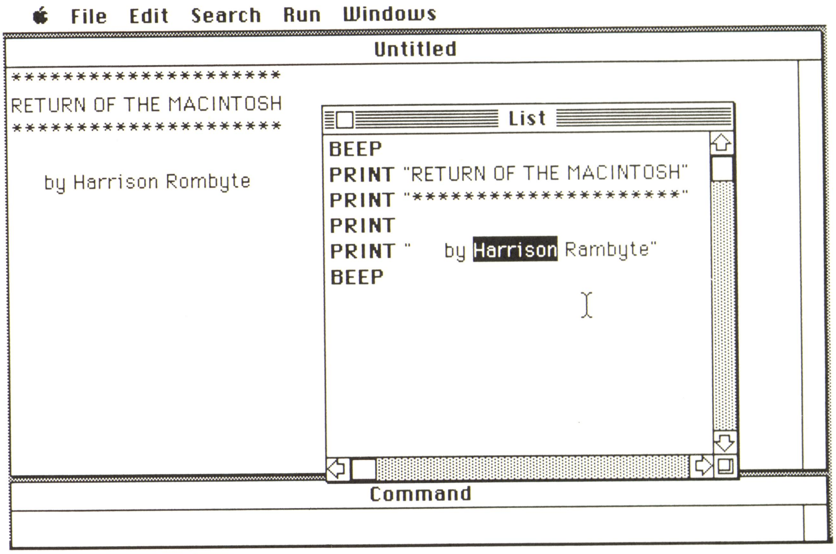
There is something else we can do with text we've selected: We can replace it with new text. The result is the same as deleting and typing something new in its place, but it saves a step.

Suppose we wanted to change the name Harrison in our program to Heinrich. We could position the insertion point just after the *n* in Harrison, backspace to the initial capital *H*, and type in the new name. But there's a faster way: We can select the text we want to delete, and then simply type what we want to replace it.

Dbl-click:	on Harrison
Type:	Heinrich

The name Harrison disappears, just as if we'd pressed the Backspace key, and the name Heinrich takes its place, as shown in Figures 3-6 and 3-7 on the following page.

The moral is that once something is selected, *anything* we type replaces it. This is a handy feature, but it can also get us into trouble if we've selected something for a reason other than to delete or replace it (which we'll do shortly), and then hit a key by mistake: Whatever was selected is then gone forever. So be careful when anything is selected.

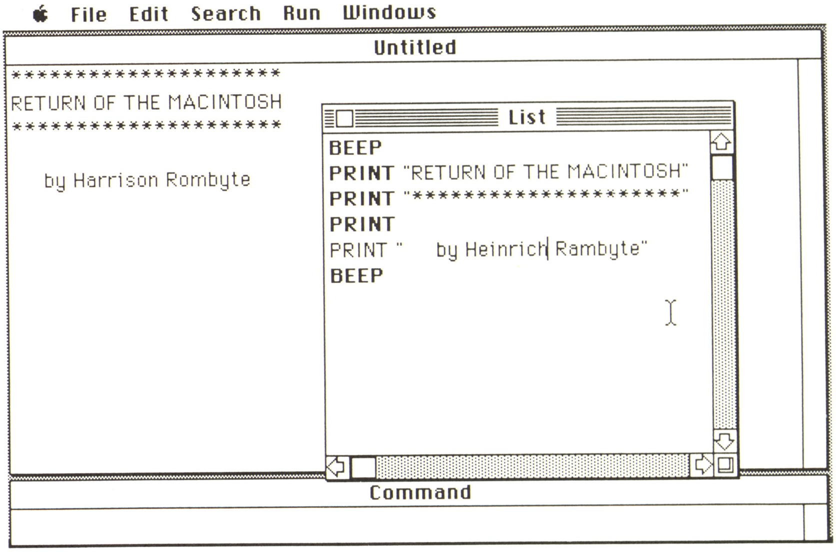

File
Edit
Search
Run
Windows
Untitled
List
Command

```
*********************
RETURN OF THE MACINTOSH
*********************

   by Harrison Rombyte
```

```
BEEP
PRINT "RETURN OF THE MACINTOSH"
PRINT "*********************"
PRINT
PRINT "     by Harrison Rambyte"
BEEP
```

FIGURE 3-6

Selecting "Harrison"


File
Edit
Search
Run
Windows
Untitled
List
Command

```
*********************
RETURN OF THE MACINTOSH
*********************

   by Harrison Rombyte
```

```
BEEP
PRINT "RETURN OF THE MACINTOSH"
PRINT "*********************"
PRINT
PRINT "     by Heinrich Rambyte"
BEEP
```

FIGURE 3-7

Program line after "Heinrich" typed with "Harrison" selected

Now that you know how to select and delete text, you no longer have to choose New from the File menu to clear the List window. Instead, you could simply drag the I-beam from the beginning to the end of the program and then press the Backspace key.

Inserting new lines of text

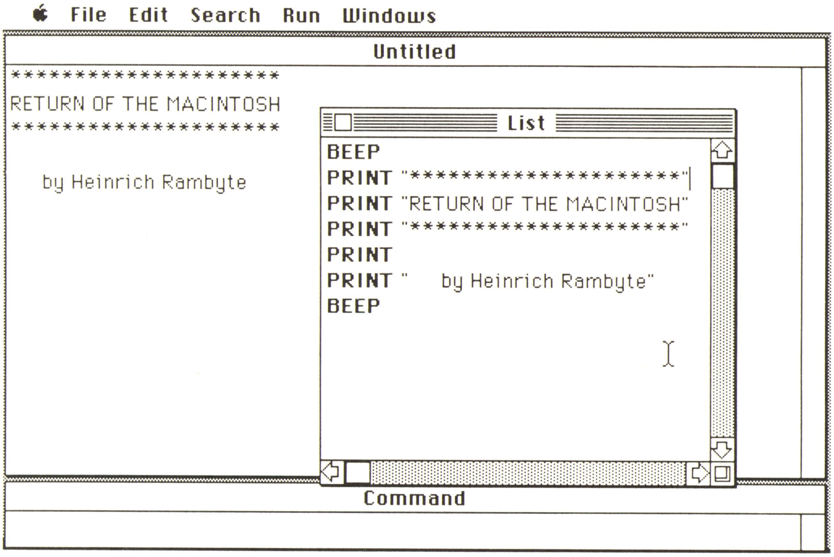
When writing a program, we will quite often need to insert a new program line between two existing lines. There are two ways to do this. One way is to position the insertion point at the beginning of the line *just below* where we want the new line to go, then press Return. This moves that line down and creates a new blank line above it. To put text on this blank line, move the insertion point up to the beginning of the line and start typing.

A second, more straightforward method of inserting a new, blank line is to simply set the insertion point at the end of the line *above* the one we want to insert and press Return. As an example, let's retype the line of asterisks we deleted from our program.

Click: the I-beam pointer after the first BEEP statement
Press: [Return]
Type: print “*********************”

The List window should now look like Figure 3-8. To display the results in the Output window:

Choose: Start from the Run menu



```
*********************
RETURN OF THE MACINTOSH
*********************

   by Heinrich Rambyte
```

```
BEEP
PRINT "*********************"
PRINT "RETURN OF THE MACINTOSH"
PRINT "*********************"
PRINT
PRINT "    by Heinrich Rambyte"
BEEP
```

FIGURE 3-8

The program with asterisks restored

We can do most of our editing, especially on short programs, by using the backspace and insertion techniques just described. However, as our programs grow longer, we're going to want to call upon the powerful editing commands that are available under the Edit menu. Let's take a look at them now.

The Edit menu

The purpose of the Cut, Copy, and Paste commands in the Edit menu is to move sections of a program from one place to another, and to duplicate parts of a program. In a program with a small number of lines, this may not seem like a very important feature. But if we're writing a longer program and find, for instance, that we'd like to move 20 program lines from the top of the program to the bottom, then we'll be very happy that we know how to cut, copy, and paste.

The Cut command

The Cut command does just what its name implies: It deletes a selected chunk from a program listing and stores it temporarily on the *Clipboard* (which we'll discuss shortly). This chunk can be either a program line, part of a line, or a number of lines. We select whatever we want by dragging the I-beam pointer across it, as we did earlier. Let's use Cut to delete the first line of asterisks again.

Select: PRINT "*********************"
Choose: Cut from the Edit menu

At this point the line disappears, leaving a blank line in its place. To close up the lines:

Press: [Backspace]

The List window now looks like it did when we deleted the line with the Backspace key. However, there is an important difference: When we delete something from a program with the Backspace key, it's gone forever, but when we delete something using the Cut command, it's saved on the Clipboard. What is the Clipboard?

The Clipboard

The Clipboard is a part of the Mac's memory where selected text can be placed temporarily with the Cut or Copy commands. Whatever is currently on the Clipboard can be retrieved with the Paste command, as we'll see shortly.

The amount of text we can put on the Clipboard depends on how much memory we're using to hold other things, like our program. But the programs in this book are short enough that we should be able to put anything we want on the Clipboard.

The important thing to remember about the Clipboard is that it can only hold *one* cut or copied section at a time. So if we cut something, it goes on the Clipboard; if we then cut something else, the first section is lost, since the Clipboard can hold only one section of text at any one time.

Right now, the line we cut from our program is on the Clipboard. How do we get it back if we need it?

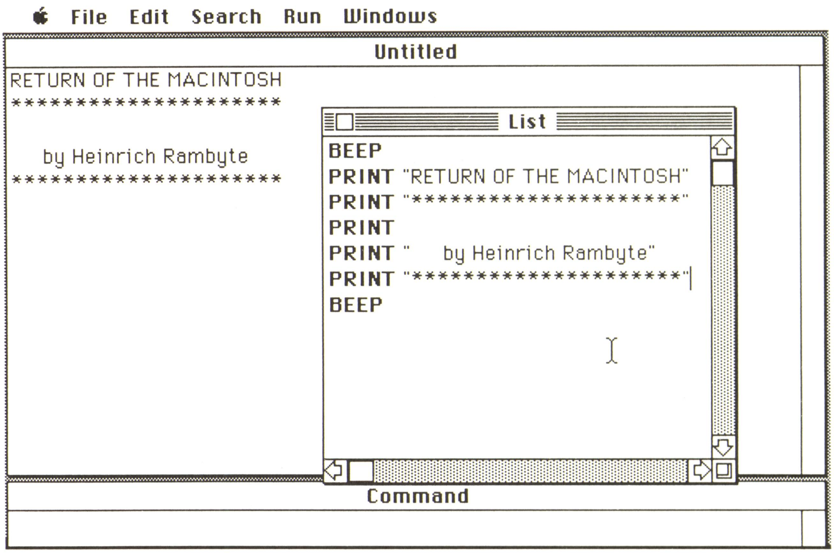
The Paste command

The selection on the Clipboard can be brought back into a program with the Paste command, and the cut material is pasted starting *at the current insertion point.* Since we can put the insertion point wherever we want, we can also paste the Clipboard contents wherever we want. And, until we cut or copy something else to the Clipboard, we can paste the current contents as often as we want wherever we want in the program, because pasting does not erase the contents of the Clipboard.

Let's paste the line we just cut back into our program, but at a new location. How about just below our fictitious programmer's name?

Click: the I-beam pointer at the end of
PRINT "   Heinrich Rambyte"
Press: [Return]
Choose: Paste from the Edit menu

Presto! The PRINT statement with its asterisks is written from the Clipboard back into our program following the insertion point. Now



```
RETURN OF THE MACINTOSH
***********************

   by Heinrich Rambyte
***********************
```

```
BEEP
PRINT "RETURN OF THE MACINTOSH"
PRINT "***********************"
PRINT
PRINT "   by Heinrich Rambyte"
PRINT "***********************"
BEEP
```

FIGURE 3-9

The program with the new line pasted in

when we run the program, our screen looks like Figure 3-9. We can also cut and paste parts of a line, or more than one line, using this technique.

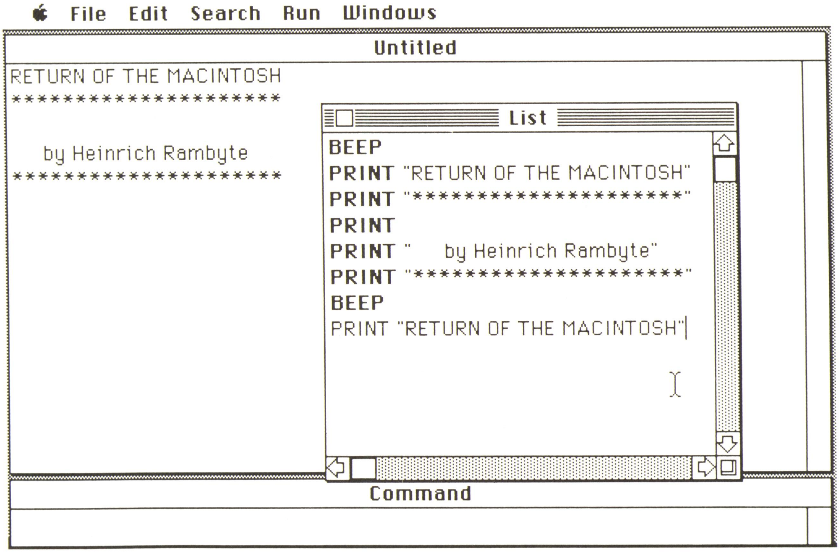
The Copy command

The purpose of the Copy command is to let us *duplicate* parts of our program. It's a little like Cut in that it takes the part of our program that we've selected and puts it on the Clipboard. But unlike Cut, it *doesn't delete* the selected text from our program. Instead of deleting the selection and putting it on the Clipboard, it puts a copy of it on the Clipboard. From there, we can use the Paste command to transfer the copy from the Clipboard to as many places in the program as we want.

As an example, let's copy the line in the program that prints the title *RETURN OF THE MACINTOSH,* then paste the copy at the very end of our program.

Select:	PRINT "RETURN OF THE MACINTOSH"
Choose:	Copy from the Edit menu
Click:	the I-beam pointer below the second BEEP statement
Choose:	Paste from the Edit menu

The line should now appear in both places, as shown in Figure 3-10.



```
RETURN OF THE MACINTOSH
*********************

     by Heinrich Rambyte
*********************
```

```
BEEP
PRINT "RETURN OF THE MACINTOSH"
PRINT "*********************"
PRINT
PRINT "     by Heinrich Rambyte"
PRINT "*********************"
BEEP
PRINT "RETURN OF THE MACINTOSH"
```

FIGURE 3-10

The result of copying and pasting the first PRINT statement

You now know everything you need to know in order to edit programs. At this point you may not feel completely confident about all these editing techniques, but as we write more programs you will become more comfortable with them, and eventually their use will seem like second nature to you.

Before we continue, let's backspace over the line we just duplicated, so we can work with our original program.

SAVING PROGRAMS

Once we've written a program, it becomes like anything else we've put time and effort into: a poem we've written, a painting we've created, or an old car we've restored. We want to keep these things from being damaged or lost; they become almost precious. It's the same with a program. Sure, this may not be so true for our little example program, but it is very important for longer programs that may have taken us hours or even days or weeks to create.

Understand that a program is a particularly fragile creation. In fact, the program now sitting in the List window can be destroyed forever simply by turning off the Mac! It will also be destroyed if there is a power failure, or if your dog runs into the Mac's power cord and pulls it out of the wall, or if a nasty "friend" selects the entire program and presses the Backspace key. Once a program is destroyed like this it's gone forever—there aren't even any torn shreds left.

How can we record the program in a more permanent way, so it will not be so dependent on the power supply and the temperament of our dog? We can save it on the BASIC work disk. This is a process much like recording a song on your tape recorder. The symbols that make up the program are written on the disk's magnetic surface in such a way that they can later be loaded back into the List window. We'll explain how to save a program and how to load it back from the disk in the following sections.

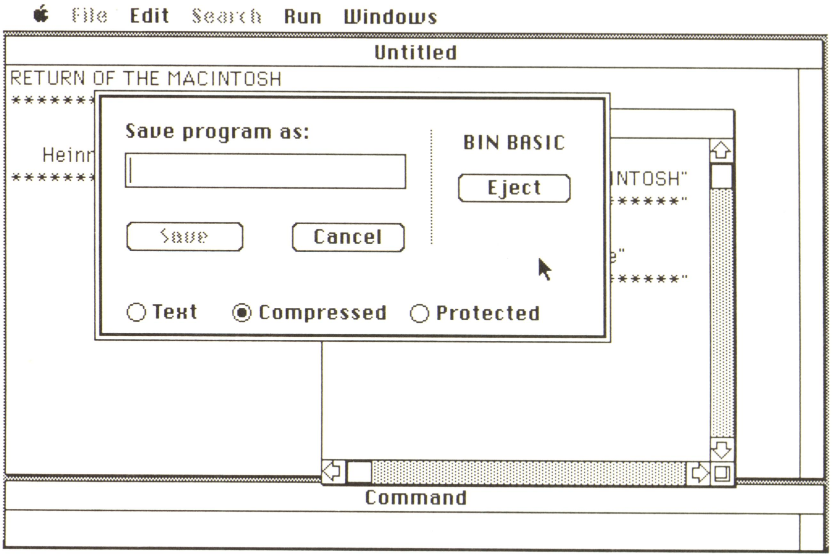
Saving our program on the disk

We save programs on the disk using one of two commands found on the File menu. Let's try one of them now.

Choose: Save from the File menu

We're presented with the dialog box shown in Figure 3-11. At the top right corner of this box is the name of our disk, and in the top left corner is the instruction *Save program as:*. Below this is a box with a blinking insertion point where we can type the name we want to give our program.

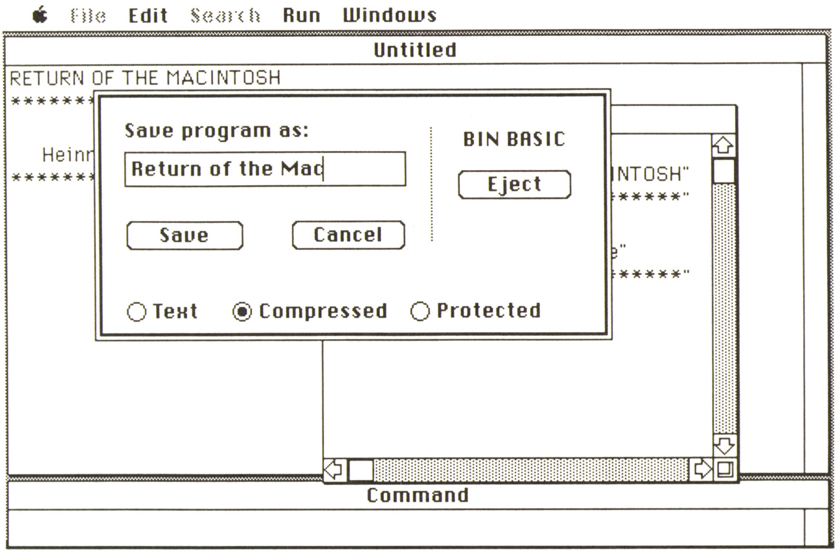
What name can we use? Any name that will fit in the space provided. The name can have spaces, use upper- and lowercase letters, and


File
Edit
Search
Run
Windows
Untitled
RETURN OF THE MACINTOSH
****
Heinr
****
Save program as:
BIN BASIC
Eject
Save
Cancel
INTOSH"
*****"
e"
*****"
Command

○ Text ● Compressed ○ Protected

FIGURE 3-11

The Save dialog box



RETURN OF THE MACINTOSH

Save program as: Return of the Mac

○ Text ● Compressed ○ Protected

FIGURE 3-12

The Save dialog box with the program name

contain punctuation such as periods and slashes (but *not* colons). Let's type a name, but don't press Return just yet:

Type: Return of the Mac

Now your screen appears as shown in Figure 3-12.

The saving process begins either when we press Return, or when we click the Save button. Let's use the button approach:

Click: the Save button

The disk whirs as the Mac writes our program on the disk. Now our program is safe because it's stored on the disk as a *file,* and it has its own *file icon* on the Mac desktop. We can turn off the Mac, or the power can fail, and our program will still be available on the disk. (Of course, if you lose the disk or run over it with your car the program will be lost, but we can't help you with that!)

One thing to notice is that the title in the title bar of the Output window has changed from *Untitled* to *Return of the Mac.* The Output window is always titled with the name of the most recently saved version of the program.

Now that we've saved our program, let's take a look at the other buttons in the Save dialog box and see what they can do for us.

Other options when saving your file

The other buttons in the Save dialog box offer a number of different options when saving our file to the disk. You don't really need to know about all of them now, but we'll discuss them briefly in case you should need them in the future.

Canceling the Save

Suppose we call up the Save dialog box and then decide that we don't want to save the program after all. We can close the box and return to the BASIC desktop by clicking the Cancel button. When we click Cancel, the Save dialog box disappears and we return to the BASIC desktop.

Changing drives and disks

If you have two disk drives, you may want to save your program on a disk in the external drive (assuming your BASIC disk is in the internal drive). If a second drive is connected to your Mac you will see another button, called Drive, in the Save box just below the Eject button. If there's no disk in the drive the button is inactive (dimmed); the button becomes active when a disk is inserted.

We can tell BASIC we want to change drives by clicking on this Drive button. The name of the disk in the upper right corner of the box changes, indicating that anything we save will be saved on this disk. We can switch back and forth between the external and internal drives by clicking on the Drive button.

We can also eject the disk in the currently selected drive by clicking the Eject button, insert a different disk, then click the Save button and save the program on that disk. If you're using a single-drive Mac, you can do the same thing, but you'll have to swap the BASIC disk and the second disk a few times.

The ability to save and load programs on disks other than the BASIC disk means that we can write long programs, and that we can store many more programs than we could if we were confined to the BASIC disk, which already has a lot of space taken up with the BASIC program itself, and the system folder.

Saving the program as Compressed

At the bottom of the Save dialog box you'll see three *radio buttons* labeled Text, Compressed, and Protected. They are called radio buttons because, like the buttons on your car radio, only one of them can be active at any one time. The button that is active has a black dot in the middle.

The option that is normally active is Compressed. This tells BASIC to save your program on the disk in a special compressed format, which takes up less space on the disk.

Saving the program as Text

Later on, as you gain programming experience, there will be times when you won't want to compress a program but will instead want to save it as a simple string of letters and numbers, called a *text file*. This is necessary in certain situations, such as when we want to send a program over the phone lines with a modem, or copy it into a MacWrite or other word-processing file.

When we need to save a program as a text file, we simply click the Text radio button. Then, when we save the program, it is saved in a non-compressed text format.

Saving the program as Protected

It may happen that you will write a program that you want people to use, but you don't want them to see the listing of the program or modify it. Clicking the Protected radio button before we save our program has just this effect. The program is saved as a *protected file*. We can run a protected program, but neither we nor anyone else can use the List window to look at the program, and we can't modify it.

There is an obvious danger here. If we protect our program by mistake, we will end up with a program we can no longer look at or modify. If this is the only copy of a program we were in the middle of writing, this could be very inconvenient. So make sure you mean it when you click the Protected radio button.

Saving your program again... and again

Suppose we make some improvements in our program, and then want to save the new version on the disk, replacing the version we saved before. This is a common situation when we are developing a program: We'll write some program lines, save the program, write some more lines, save the program, change some lines, save the program, and so on. But how do we save a program that's already been saved? It turns out this is very easy.

Choose: Save from the File menu

That's it. No Save dialog box appears, since BASIC already knows the name of the program. The version of the program currently in the List window is saved under the name in the title bar (*Return of the Mac* for the example above), and the old version on the disk is lost. So be careful: If you save the program after making several changes, you won't be able to get the old version back.

But suppose we write a version of our program that we're happy with, and then make some modifications to it, creating a new version that we also like. So we want to save the new version, but we want to keep the old version as well. How can we save a copy of the new version of the program on the disk, without destroying the old one? The answer lies with the Save As... command.

The Save As... command

The Save As... command, which is directly below the Save command in the File menu, was created to let us save different versions of a file under different names. Instead of automatically saving our program under the old name, which destroys the old version on the disk, choosing Save As... brings back the Save dialog box we just worked with. But now, the name of the current file is already displayed in the name box, highlighted in black. If we click the Save button, the program is saved under the current name, just as if we'd chosen Save from the File menu. But if we start typing a new name, the highlighted name immediately disappears. When we finish typing the name and click the Save button, the current version is saved with that name, leaving the original intact. From now on, we can load and work with either version of the program, as we'll discover in the next session.

LOADING A PROGRAM FROM THE DISK

Now that our program is safely stored on the disk, let's clear the desktop and experiment with loading it.

Choose: New from the File menu

The New command causes the program to be erased from memory, the listing of the program to disappear from the List window, and any output that remained in the Output window to be cleared away, leaving us with a cleared desktop. How do we get our program back?

Choose: Open... from the File menu

The dialog box shown in Figure 3-13 appears.

This box contains a small window that shows the names of all the BASIC programs that are stored on the disk—in this case, only the one we just saved. But there can be more program names than will fit in the file-name window at one time. When there are, the scroll bar on the right side of the name window will be active, and we can use it to move the list of programs up and down.

Notice that the complete name of our program is not listed in the window. BASIC truncates long names and includes ellipses to indicate which names it has cut short. (This is a good reason for keeping program names short.)

Click: Return of the ...

The name becomes highlighted, indicating that it's selected. At the same time, the Open button becomes active. We're now ready to open the file.

Click: the Open button

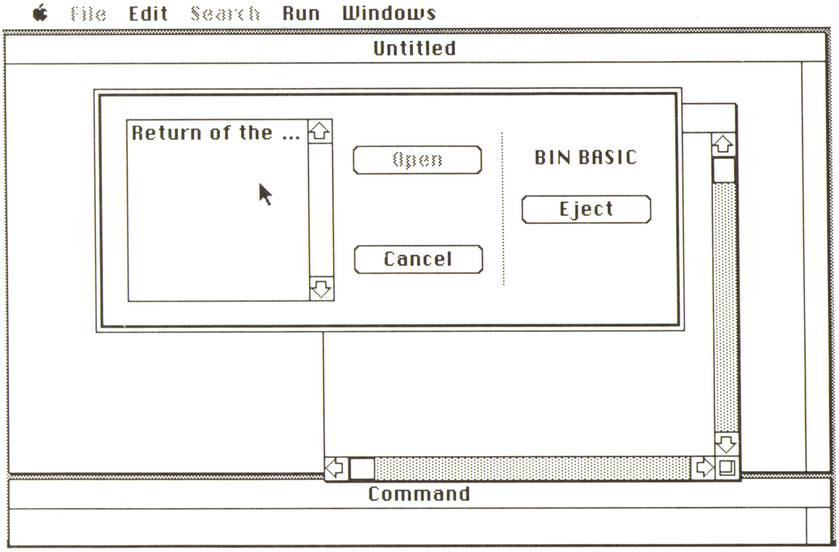


FIGURE 3-13

The Open dialog box

The disk drive whirs again, and our program appears in the List window. It has been loaded from the disk into memory. We can now run it, modify it, and save it on the disk again in the ways described above.

(Before we leave the Open command, we should point out that there's a shortcut for opening a file: Simply double-click in the name window on the name of the file that you want to open.)

Other options in the Open dialog box

As in the Save dialog box, we can click the Cancel button if we decide we don't want to open a program after all. If the program we want to open is on a different disk, we can click the Eject button. When we insert our new disk in the drive, its name appears just above the Eject button, and the program names in the program-list window change to those on the new disk. If you have a second drive, you'll also see a Drive button that permits you to change back and forth from one drive to another.

In the following chapters we'll have many opportunities to practice these saving and loading operations, so don't worry if you don't feel entirely at ease with them at this point.

QUITTING FROM BASIC

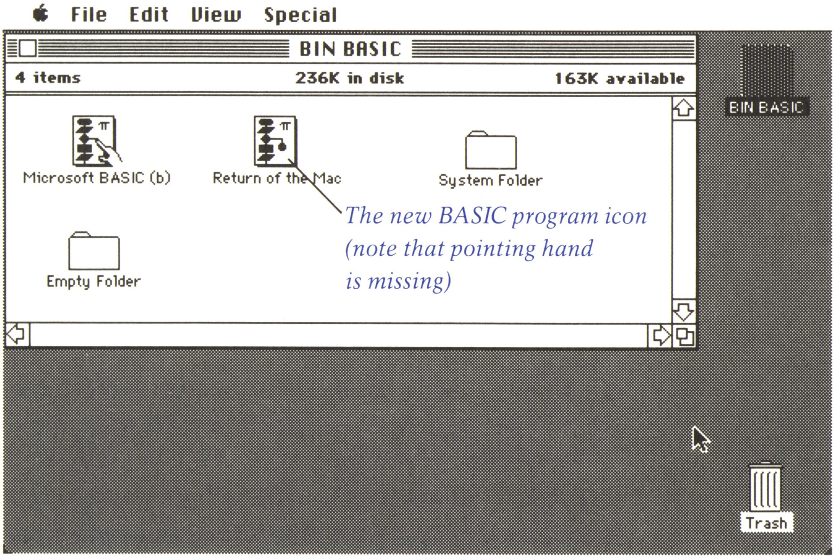
Suppose we've been working on our program and we decide to quit for the day and turn off our Mac. What do we do?

First, we make sure that we've saved the most current version of our program, as we just did. Then we need to get out of BASIC and back to the Macintosh desktop, so we can eject our disk and turn off the power.

Choose: Quit from the File menu

The BASIC desktop disappears, and in a few seconds we're back on the Macintosh desktop, as shown in Figure 3-14.

The desktop looks almost the same as the way it was when we started, with one exception: There's a new icon visible—our *Return of the Mac* program that we saved to the disk (we've also rearranged the desktop a bit for clarity). You can see that it is similar to, but not exactly like, the icon for Microsoft BASIC: Only the pointing hand is missing. This icon is called a BASIC *program* icon, to distinguish it from the Microsoft BASIC


File
Edit
View
Special
BIN BASIC
4 items
236K in disk
163K available
Microsoft BASIC (b)
Return of the Mac
System Folder
Empty Folder
BIN BASIC
Trash

The new BASIC program icon (note that pointing hand is missing)

FIGURE 3-14

The Macintosh desktop, with a new icon

application icon we have been using. All program icons that we create with BASIC look this way (except that they also differ according to which version is used, binary or decimal).

STARTING A PROGRAM FROM THE MACINTOSH DESKTOP

We described earlier how to load a program from the disk by choosing Open from the File menu on the BASIC desktop. We can also load a program directly from the Macintosh desktop. Before we end this chapter, let's see how this is done. Then you'll be due for a well-earned rest.

Dbl-click:	the Return of the Mac icon

The disk whirs, the Mac desktop disappears, and suddenly our program executes: The title appears on the screen, and the Mac beeps, just as it does when we choose Start from the Run menu.

Notice, however, that nothing appears in the List window. BASIC makes the assumption that when we double-click on a program's icon, we want it to actually do what it's supposed to do, and that we're not interested in modifying it or looking at the listing. However, once the program has finished executing, we can choose Show List from the Windows menu. This will display the program listing, which we can then edit and modify.

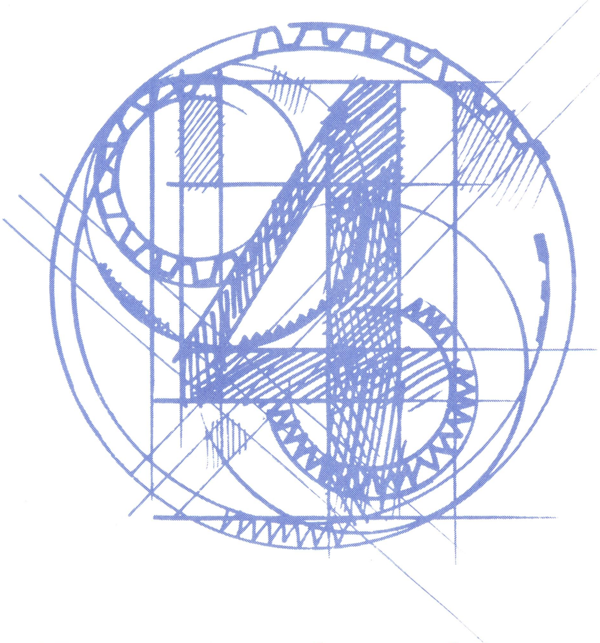
REFRESH-YOURSELF WINDOW

At this point you may not be an expert programmer, but you know almost everything you're going to need to know about handling programs themselves. You know what programs are and how to type them into the List window, and you know how to run, or execute, them. You also know how to edit a program: how to delete and insert material, and how to move material around in the program using Cut, Copy, and Paste.

You've also become familiar with using the disk to store your programs. You know how to save a program to the disk under a particular name, and how to load it back into the Mac's memory.

A fringe benefit of this knowledge is that you can type in and execute programs you find printed in books and magazines. You don't need to be a programmer to do this; all you need to know is the information contained in this chapter.

However, to learn how to write your own programs, you'll need to learn more about programming itself: how programs are constructed, and how the various statements work. In the next chapter we'll continue our journey into Microsoft BASIC programming by introducing you to one of the most exciting aspects of programming: graphics.



Introducing Graphics

Move up close to your Macintosh screen and carefully examine the various images on it. Look closely at the objects, such as icons and scroll bars. Notice the fine detail of the icons, and the letters of their names.

The images on the desktop—the icons, menus, and even the pointer—are all examples of the fine degree of visual precision that the Mac allows. This fine degree of graphic detail is also available to your Microsoft BASIC programs! Microsoft BASIC contains statements that let us draw dots, lines, and simple shapes with relative ease. This chapter will explore the topic of graphics and show you how images are drawn on the Macintosh and displayed in the BASIC Output window.

We are learning graphics now for several reasons. One reason is that we can demonstrate BASIC statements "through" graphics, thereby making them easier to visualize, and more immediate in feedback. Another reason is that the Macintosh *is* a graphics environment. Any attempt to program the Mac must utilize this graphics environment to live up to the Macintosh standard. Furthermore, statements you have already learned, such as PRINT, have further abilities related to the graphics environment. For example, up to now we have only PRINTed things starting at the left edge of the screen, and on a line BASIC has picked for us. We will see that it is possible to PRINT anywhere at all on the screen. We will also learn about drawing dots, lines, boxes, and circles. However, we will not cover the more sophisticated QuickDraw graphics routines in this chapter. Since QuickDraw is a rather large topic, we have devoted an entire chapter—Chapter 15—to it.

PIXELS: THE BUILDING BLOCKS OF GRAPHICS

If you view the Macintosh desktop up close you will notice that certain diagonal lines seem to be made up of little dots. These little blips are the secret of how the Macintosh's designers made their fortune. Simply said, *every object, every letter, every number, everything or anything displayed on the Mac's screen is made up of dots.* Because of the way the Mac was designed, you can't see these dots when they are side by side in perfect vertical or horizontal lines, but lines on an angle show them clearly.

The dots are so small that you can't easily tell how they make up an object. "Dots" is a rather poor name for these blips. What is really going on is that the computer is forcing a tiny part of the screen to turn black—it is affecting a *picture element.* These words have been condensed over the years by computer scientists to form a new word: *pixel.* A pixel is a single dot. On the Mac, a pixel can be white or black. You may be familiar with pixels if you have used FatBits in MacPaint. FatBits causes a portion of a MacPaint drawing to expand on the screen, and then the pixels appear as little black squares.

THE COORDINATE SYSTEM

How are these pixels organized on the screen? Imagine that the Macintosh screen is divided up into a flat grid, like one found in a screen door. The grid is made up of 512 columns and 342 rows as shown in Figure 4-1, which makes a total of 175,104 pixels.

Each pixel occupies a fixed location on this grid. Much like a map of city streets, a pixel is found at the intersection of a vertical column and a horizontal row. There are 512 columns from left to right, and 342 rows of pixels from top to bottom. Note that this grid is sequenced in "computer number" style: The rows and columns start at 0, rather than at 1. The rows extend from 0 to 341, and the columns from 0 to 511. Also note that the top row starts with 0, so the numbers of the rows get larger as we go down the screen. It might seem upside down to number the top row 0 and the bottom row 341, but that is the way it has been done for years by Apple and IBM. Numbering the columns from 0 on the left to 511 on the right seems more natural.

BASIC has a slightly different way of locating pixels on the screen. Instead of referring to rows and columns, BASIC refers to the *x and y coordinates* of a pixel. The x coordinate in BASIC refers to the *column* the pixel is in, and the y coordinate refers to the *row* the pixel is in. So in BASIC statements, we locate a specific pixel on the screen by specifying its coordinates as a pair of numbers. For example, in Figure 4-1, a pixel is shown at location (5,4). Thus the first number (x coordinate) describes how far over the pixel is from the left side of the screen, while the second number (y coordinate) describes how far down the screen the pixel is positioned.

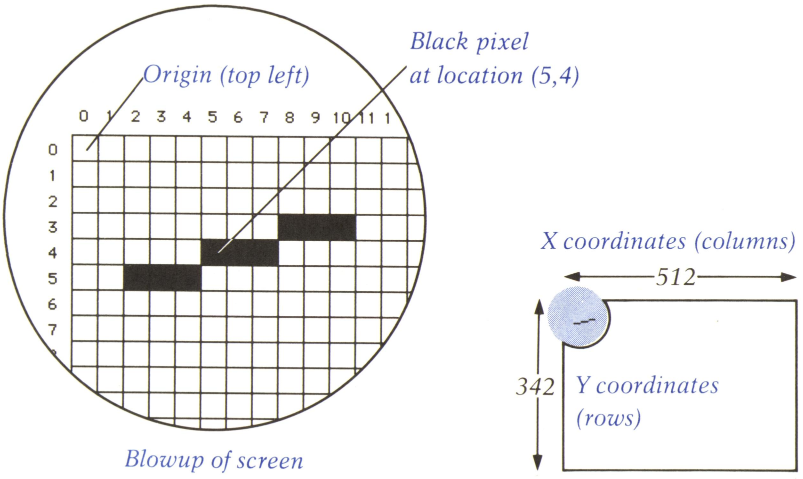


FIGURE 4-1

The Macintosh screen is made up of pixels arranged in 512 columns and 342 rows.

As reference points, the *top left* corner of the screen is (0,0) and is referred to as the *origin*. Technically, the *bottom right* corner of the screen is located at (511,341).

BASIC desktop vs. Output window

Before learning how to control the thousands of pixels on the screen, you need to understand the way the coordinate system is organized in the BASIC environment. BASIC does not let us control all the pixels, but only those in the Output window.

Figure 4-2 shows how the coordinate system for BASIC is organized. As you can see, the origin coordinate (0,0) is now at the top left corner of the *Output window*. The Output window we're given when BASIC first starts up does not extend the full 512 columns and 342 rows we described, but only 490 columns and 253 rows. The reason is that the desktop's menu bar, the Output window's title bar, and the Command window consume 89 rows of pixels from the screen's total of 342 rows, and the right scroll bar consumes 22 columns from the total 512 columns from the Output window. Can we still use values up to 512 and 342? We can, but since these coordinates are hidden behind the menu, title, and scroll bars, they will not show up in the Output window.

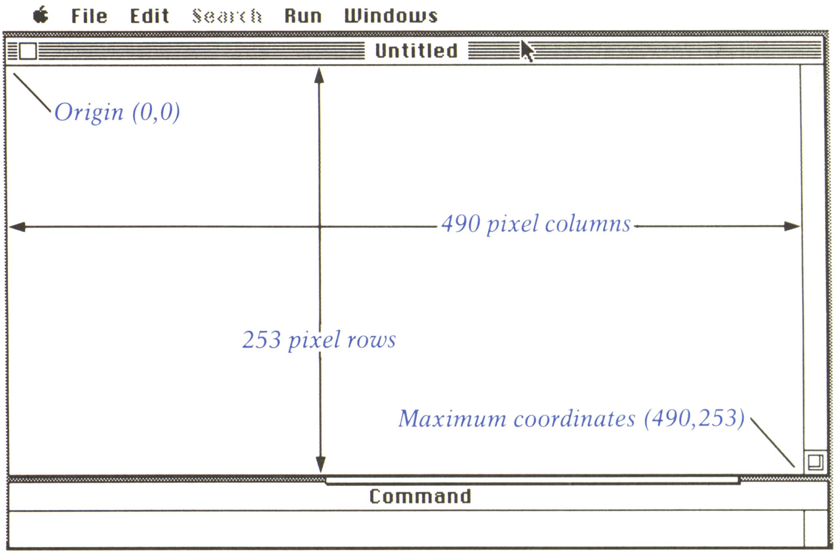


FIGURE 4-2

Coordinate system on the Output window

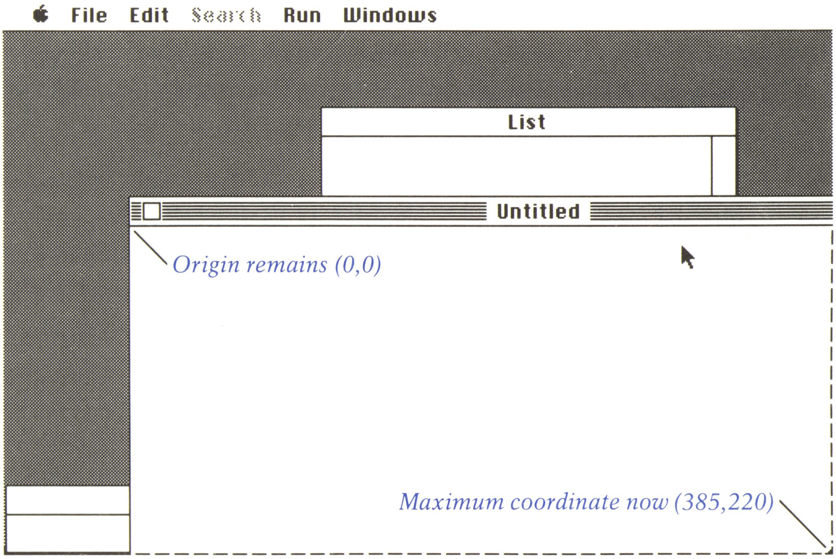


FIGURE 4-3

Moving Output window reduces maximum visible coordinates

Pixel coordinates and the Output window

Figure 4-3 shows what happens to the coordinate system when we move the Output window. The origin (0,0) follows the Output window wherever we move it. However, more of the window is invisible, and therefore the coordinates for the *visible* lower right corner are reduced. Furthermore, the maximum bottom right coordinates are also readjusted whenever the Output window is resized. The point to remember is that the Output window has its own coordinate system and that *it is relative to the top left corner of the window.*

PLOTTING WITH PIXELS: PSET

Now that we know that the screen is covered with thousands of little pixels, we need a mechanism that allows us to move to any pixel we want and make it black or white. Moreover, wouldn't it be nice if we could do this repeatedly to form letters, shapes, and other recognizable items? Well, we can: BASIC's PSET statement allows us to control any one of the more than 175,000 pixels on the screen.

The P of PSET stands for *point* because this statement puts a point on the screen where we want it. You can also think of the P in PSET as meaning to *plot* a pixel. *Plotting* is a word coined by mathematicians to describe putting points on a graph. Let's see how it works.

First, make sure you have a visible Output window and that the Command window is active. If the List window is visible, click in its close box to close it.

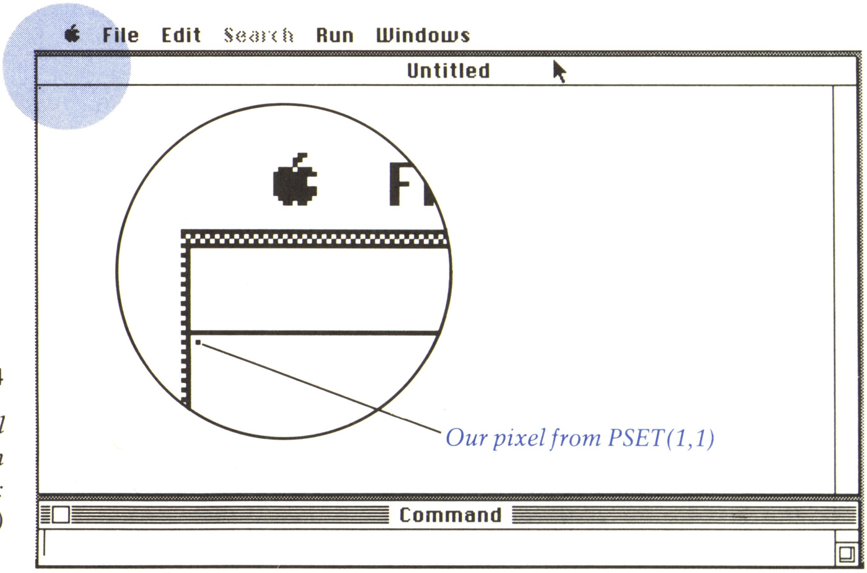
Type:	cls
Type:	pset(1,1)

Look very carefully at the upper left corner of your Output window, right below the title bar and a tiny bit to the right of the window's edge. You should see a tiny dot there, as shown in the enlarged view of the corner in Figure 4-4. (If you don't see the dot, make sure that the Output window is visible, then repeat the previous PSET instructions.) The dot, or pixel, was placed one column over and one row down from the upper left corner of the window. We can say that the dot has the coordinates (1,1).

Let's try another PSET example:

Type:	pset(489,252)

This time, look very closely at the lower right corner of the Output window. You should see a second black pixel there. This pixel is one row up from the bottom and one column from the right edge of the default setting of the Output window. We can continue to type PSET statements anywhere on the screen and each will display a black dot. But what if we want to erase a dot?



Our pixel from PSET(1,1)

FIGURE 4-4

The pixel plotted with the statement: PSET(1,1)

PSET and color

Microsoft BASIC lets us control whether a pixel is plotted in black or in white. Make sure the dot at location (1,1) is still on the screen, and if it isn't, put one there with PSET(1,1). Now in the Command window:

```
Type:      pset(1,1), 0
```

What happened? Did the black pixel disappear? It should have. The 0 at the end of the PSET statement told PSET to plot its pixel in the color white. (Note that black and white are considered colors here.) Since the background is already white, the pixel was erased. Now try this in the Command window:

```
Type:      pset(1,1), 1
```

The pixel at (1,1) reappears. You have probably concluded that the number at the end of the PSET statement controls the color of the pixel, and you're right: The number 1 equals black, and the number 0 equals white. But remember we didn't use a number when we first plotted a black pixel with PSET(1,1). That's because pixels are most often plotted as black-on-white, so BASIC makes it easier on us by not requiring a number for a black PSET. But since every color has a number in BASIC, one was assigned to black as well. The comma at the end of the PSET statement shows that the color value is optional. When we don't use a number after PSET, it defaults to black.

In Macintosh BASIC, the numbers 30 and 33 also work with PSET for white and black, and this is how they are described in the Microsoft BASIC Interpreter manual. Although either way works, we will usually call white 0 and black 1 in this book, as it's clearer. (As an aside, Microsoft BASIC for the IBM PC uses 0 and 1 for white and black, so by using 0 and 1 you will be keeping your programs compatible with the IBM PC.)

Although the Macintosh only displays the colors black and white now, its software operating system is in fact capable of handling a huge range of colors if Apple should one day release a color Macintosh. Microsoft set up its BASIC for the Mac so that it, too, could work in color. The PSET statement is capable of producing a range of up to 65,536 different colors, if Apple should ever make a Mac that shows that many. In the meantime, BASIC only understands black and white pixels.

PSET is a versatile statement. And with a lot of practice, you could learn to draw just about anything you wanted on the screen, just by using the PSET statement to make each individual dot. But drawing a picture of any complexity dot by dot would require *lots* of PSET statements. So let's look at some other BASIC graphic statements that make drawing easier.

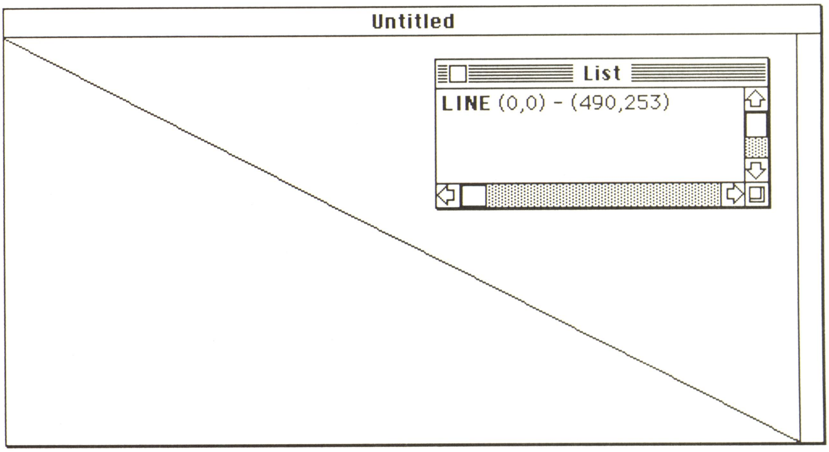
THE LINE STATEMENT

Microsoft BASIC provides a statement called LINE. Its purpose is to draw lines at any angle in the Output window. Additionally, the LINE statement has some optional features that cause it to draw boxes of any size or shape. It also has the ability to *fill* boxes with black or white.

Drawing lines with LINE

Let's see how LINE works. From now on we'll be working with the List window, so let's get the Command window out of our way by clicking its close box. Next, choose Show List from the Windows menu (or press Command-L), then resize the List window to the position shown in Figure 4-5. Enter the program line shown in the window. When it's correct, choose Start from the Run menu (or type Command-R).

What do you get? Figure 4-5 shows the result: BASIC drew a line from the top left corner of the Output window to its bottom right corner. Imagine doing that with individual PSET statements! In effect, the LINE statement condenses a series of PSET statements into one.



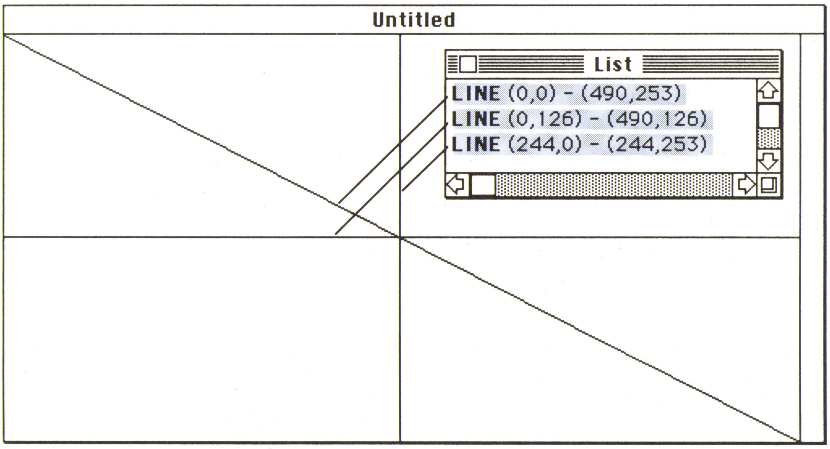
```
LINE (0,0) - (490,253)
```

FIGURE 4-5

The diagonal line drawn by the statement LINE (0,0) – (490,253)

FIGURE 4-6

A simple program using LINE statements

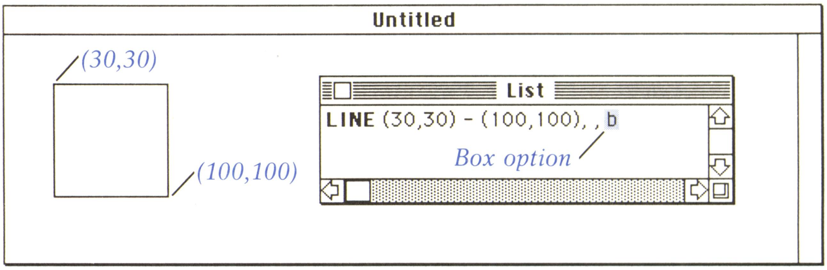


```
LINE (0,0) - (490,253)
LINE (0,126) - (490,126)
LINE (244,0) - (244,253)
```

As you've probably guessed, the first pair of numbers in parentheses following LINE give it the starting coordinates of the line we want drawn, and the second pair give it the ending coordinates. So by varying these numbers, we can draw lines at any angle that will fit in the Output window. To try this, type the three LINE statements shown in Figure 4-6, then choose Start from the Run menu (or press Command-R).

The values we have selected for the three LINE statements in this program intersect at the exact middle of the window. The fact that the line is made up of pixels and that these are arranged on a grid of limited resolution means that the line must be simulated with tiny steps. The steps are called *jaggies* in computer graphics terminology. If the jaggies are too big (depending on the angle of the line), a line doesn't appear to be a line: It looks like the edge of a crude saw. The Mac has such high resolution that most of the time a diagonal line will look fairly smooth. You might want to refer back to Figure 4-1 to see how this staircase effect looks when we blow up the pixels.

You will find many uses for lines with BASIC on the Macintosh. You can draw grids and shapes of all kinds, and even do shading with them. Future examples will show you how these things are done.



```
LINE (30,30) - (100,100), , b
```

FIGURE 4-7

The box drawn by the statement LINE (30,30) – (100,100),,b

Drawing boxes with LINE

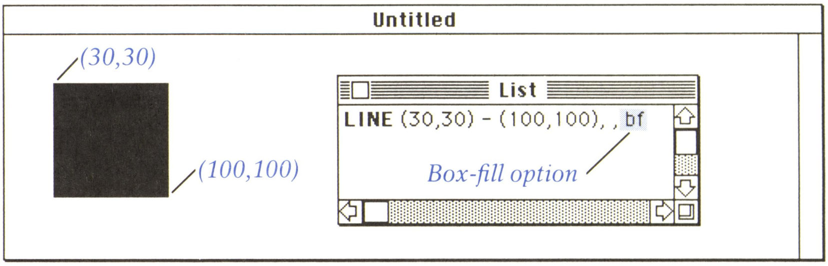
The LINE statement can also be used to draw simple boxes and rectangles. Let's clear our current program from the List window by dragging the pointer through it and pressing the Backspace key. We'll then enter and run the program shown in Figure 4-7.

A rectangle appears in the Output window! What is different in the LINE statement here is that two commas followed by the letter *b* have been added at the end (we'll explain what the commas mean soon). The *b* is an *option* signifying that LINE is to consider the two coordinate pairs as *the opposite corners of a rectangle, or box*. The first coordinate pair locates the top left corner of the rectangle on the Output window grid, and the second coordinate pair anchors the bottom right corner.

Drawing filled boxes with LINE

Not only can we draw empty boxes with a single LINE statement, we can also draw boxes filled with the color black. To see how simple this is, just add the letter *f* to the existing program line, so your option is *bf* as shown in Figure 4-8.

Now a black-filled rectangle appears in the Output window. The *bf* at the end of the statement specifies that the *b*ox be a *f*illed box, hence *bf.*



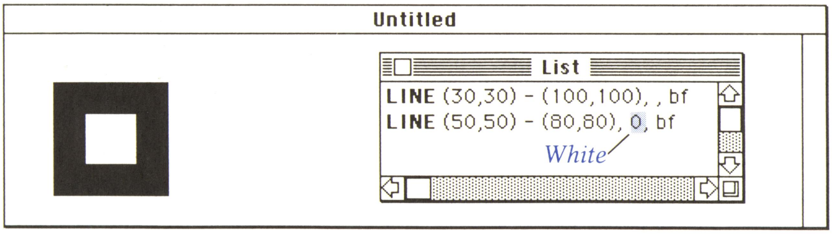
```
LINE (30,30) - (100,100), , bf
```

FIGURE 4-8

The black-filled box drawn by the statement LINE (30,30) – (100,100),,bf

FIGURE 4-9

A white rectangle drawn inside the black rectangle using white color-number option



```
LINE (30,30) - (100,100), , bf
LINE (50,50) - (80,80), 0, bf
```

We can also use LINE to draw boxes filled with the color white. First delete the current program line and then enter and run the line shown in Figure 4-9.

A white-filled box instantly appears in the middle of the black-filled box from the last example, as shown in Figure 4-9. The color option works with LINE just like with PSET: 0 (or 30) after the first comma creates white-filled boxes, and 1 (or 33) creates black-filled boxes. Leaving the number out defaults to black.

Although we've only worked with white- and black-filled boxes so far, BASIC allows us to access another set of graphics statements that can fill boxes with any shade or pattern we wish. We'll cover these in Chapter 15, QuickDraw.

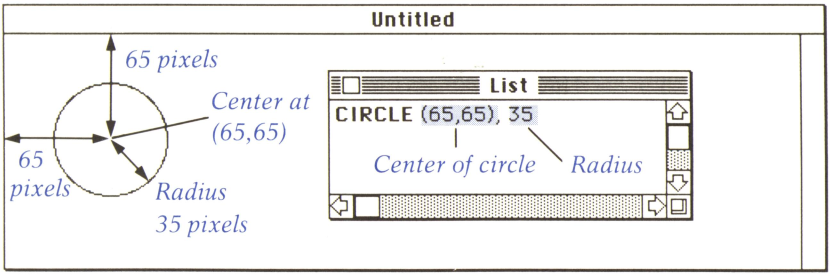
GATHER ROUND: THE CIRCLE STATEMENT

The CIRCLE statement is great for drawing sections of arcs, ovals, and curved objects. But we will only explore the ability of the statement to draw perfect circles in this book. You can learn about drawing ovals and arcs in the manual.

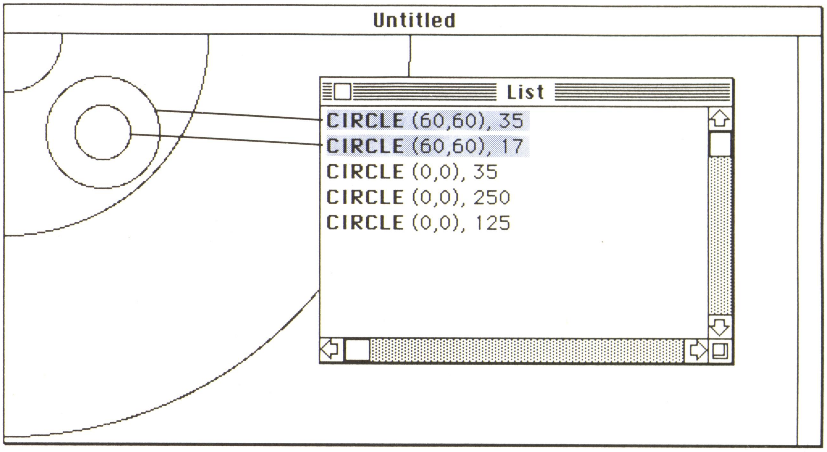
Let's try out CIRCLE by clearing the current program lines from the List window and entering and running the program shown in Figure 4-10, which produces the circle shown in the Output window.

FIGURE 4-10

The circle drawn by the statement CIRCLE (65,65),35



```
CIRCLE (65,65), 35
```



```
CIRCLE (60,60), 35
CIRCLE (60,60), 17
CIRCLE (0,0), 35
CIRCLE (0,0), 250
CIRCLE (0,0), 125
```

FIGURE 4-11

Drawing circles with different radius and center coordinates

The format of a CIRCLE statement is that the number pair in the parentheses is the coordinates of the *center* of the circle, and the first number following the center coordinates is the *radius* of the circle. Recall that radius is the distance from the center of a circle to its edge (called the *circumference*). The *diameter* of the circle is two times its radius.

We can draw circles anywhere we want on the Output window. Let's clear the List window again, then type the program shown in Figure 4-11 and choose Start from the Run menu.

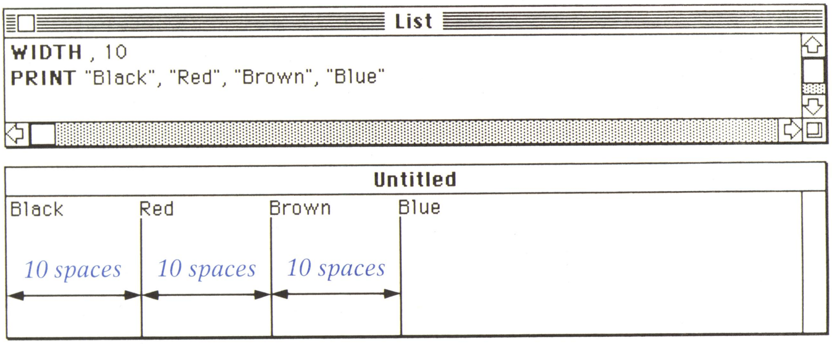
The first statement in this program draws the same size circle as before. The second statement reduces the value of the radius from 35 to 17, and draws a circle inside the first. The last three statements show what happens when we try to draw circles that are bigger than the screen, or centered so that part of the circle is drawn outside the Output window. The designers of Microsoft BASIC made it so that using graphics that are too big for the Output window doesn't create any strange side effects. Provided that the coordinates don't get *too* large, portions of graphic objects that are outside the Output window simply do not get drawn—we say that they are *clipped*.

THE TAB STATEMENT

Recall that in Chapter 2 we used commas between strings in PRINT statements to print text in specific columns on the screen. Recall also that these columns were at fixed intervals, and that the width of each column was set with the WIDTH statement, as shown in Figure 4-12.

These fixed-interval positions are similar to the tab positions that are found on a typewriter. When a comma occurs in a PRINT statement, it causes the text to drop into a column, or *print zone.* The output from a PRINT statement is displayed *left justified* in print zones. This applies to both numbers and text.

However, BASIC provides a statement that allows us to begin printing in *any* of the screen columns, rather than in just a specific zone. This statement is called TAB. To see how TAB works, let's add two lines to the program we just typed, as shown in the List window of Figure 4-13. The Output window shows the results of running the program.

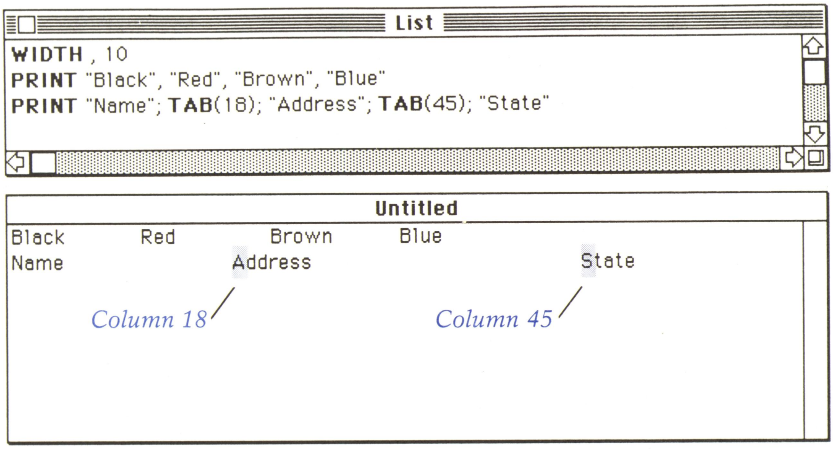


```
WIDTH , 10
PRINT "Black", "Red", "Brown", "Blue"
```

```
Black          Red            Brown          Blue
```

FIGURE 4-12

Printing in columns using the WIDTH statement and commas



```
WIDTH , 10
PRINT "Black", "Red", "Brown", "Blue"
PRINT "Name"; TAB(18); "Address"; TAB(45); "State"
```

```
Black          Red            Brown          Blue
Name                  Address                            State
```

FIGURE 4-13

Using TAB to control printing

As you can see, the string *Name* printed in the first column, then TAB(18) caused *Address* to begin in column 18, and TAB(45) caused *State* to begin in column 45. The TAB statement takes the number in parentheses and prints the text in the print column corresponding to that number.

Figure 4-14 illustrates how the Output window is organized for text. Each *0* in the figure represents a text-character *position.* As you can see, there are 16 rows for holding text, and each row can hold about 61 columns of characters (of course, if we resize the Output window there will be a different number of rows and columns). But because characters are of different widths on the Macintosh (we say the text is *proportionally spaced*), this number of columns only applies when we use just the character *0.* Furthermore, column positions referred to by BASIC's text-positioning TAB and LOCATE statements use 0 as the standard character width because it represents an average of all the possible widths.

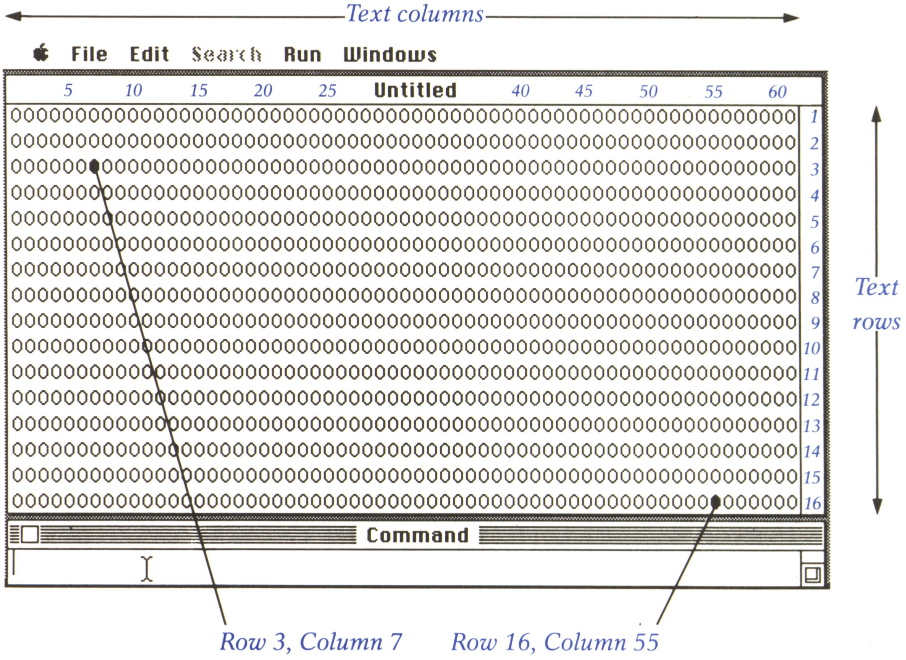
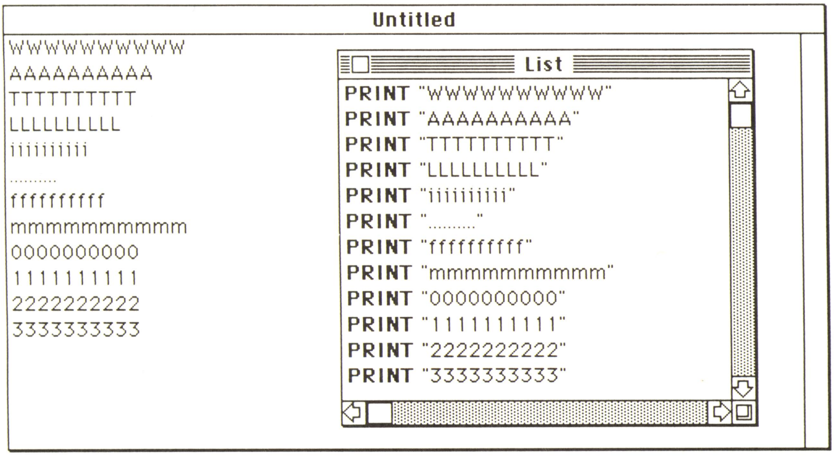

Text columns
File
Edit
Search
Run
Windows
Untitled
5
10
15
20
25
40
45
50
55
60
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Text rows
Command
Row 3, Column 7
Row 16, Column 55

FIGURE 4-14

How the Output window is organized for text

FIGURE 4-15

Different character widths for BASIC's Geneva font



```
WWWWWWWWWW
AAAAAAAAAA
TTTTTTTTTT
LLLLLLLLLL
iiiiiiiiii
..........
ffffffffff
mmmmmmmmmm
0000000000
1111111111
2222222222
3333333333
```

```
PRINT "WWWWWWWWWW"
PRINT "AAAAAAAAAA"
PRINT "TTTTTTTTTT"
PRINT "LLLLLLLLLL"
PRINT "iiiiiiiiii"
PRINT ".........."
PRINT "ffffffffff"
PRINT "mmmmmmmmmm"
PRINT "0000000000"
PRINT "1111111111"
PRINT "2222222222"
PRINT "3333333333"
```

Microsoft BASIC uses the Geneva font for its default List window and Output window text (we'll learn all about fonts in Chapter 15, QuickDraw). Geneva is a proportional font and the fact that letters and numbers are proportionally spaced and not all the same width can lead to confusion when you try to place text in certain screen locations. To understand how text characters of different widths affect the display, look at Figure 4-15. Each PRINT statement in the figure contains a string of exactly 10 characters, yet notice the radically different line lengths. You can see that the lowercase *m* and uppercase *w* are really wide, while the period and lowercase *i* are really narrow. Numbers in the Geneva font do not vary as the letters do, as the comparison of *0* and *1* shows.

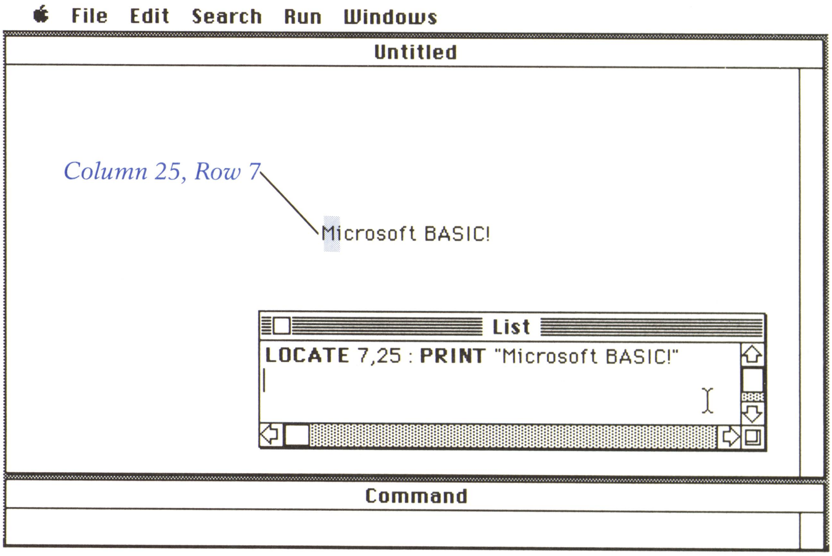
In Microsoft BASIC, we refer to a character position by its *row and column.* The TAB statement can easily move us to any *column,* but is there any way to move to a specific row? You might quickly think of using multiple blank PRINT statements. For example, the line of statements PRINT : PRINT : PRINT would move us down to the fourth row. But there is a more efficient way: the LOCATE statement.

THE LOCATE STATEMENT

Suppose you want to print "Microsoft BASIC" in the approximate middle of the Output window, say on row 7 and column 25. Rather than using a bunch of PRINT and TAB statements, we can simply use the LOCATE statement to start the text at row 7, column 25. Let's clear the List window by dragging the pointer through the program lines and pressing Backspace, then type the program shown in Figure 4-16.

What LOCATE does is move to a specific row and column relative to the top left corner of the Output window according to the two numbers that follow it: The first number represents the row, and the second number represents the column.

LOCATE is particularly handy when you want to print text in any position and in any order on the screen. To see this, resize the List window as shown in Figure 4-17, then type and run the program it contains.



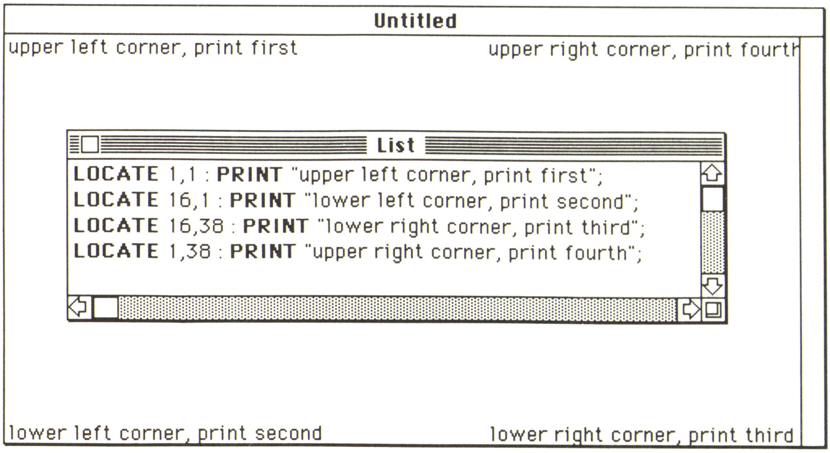
Column 25, Row 7

```
Microsoft BASIC!
```

```
LOCATE 7,25 : PRINT "Microsoft BASIC!"
```

FIGURE 4-16

The results of using LOCATE



```
LOCATE 1,1 : PRINT "upper left corner, print first";
LOCATE 16,1 : PRINT "lower left corner, print second";
LOCATE 16,38 : PRINT "lower right corner, print third";
LOCATE 1,38 : PRINT "upper right corner, print fourth";
```

```
upper left corner, print first          upper right corner, print fourth

lower left corner, print second         lower right corner, print third
```

FIGURE 4-17

Printing text at different locations with LOCATE

Each LOCATE statement causes its string to be printed in its respective corner of the screen. You could print these four lines in any order and you would always get the display in Figure 4-17. Note that we put a semicolon at the end of each PRINT string. Otherwise the screen would scroll. Recall that a plain PRINT statement causes a carriage return and line feed to occur. The semicolon suppresses this.

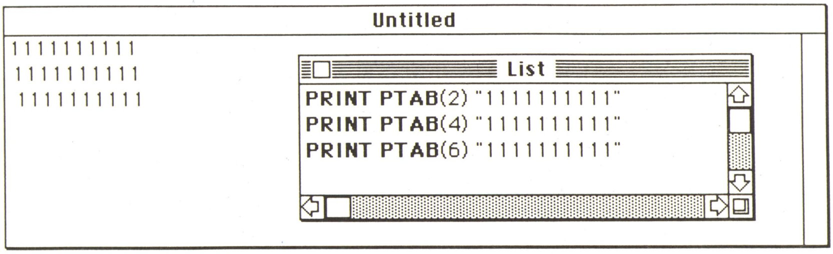
THE PTAB STATEMENT

While LOCATE lets us move to a particular row and column on the Output window, BASIC contains the PTAB statement for tabbing to any *pixel* position on a line. To see how PTAB works, let's enter and run the program shown in Figure 4-18.

Note that each successive string was shifted two pixels to the right before it was printed out. Regardless of which row we are on, PTAB moves to the pixel position *in that row* specified by the number in parentheses. For example, PTAB(2) moves the beginning of text to the second pixel position. PTAB(4) moves us to the fourth pixel position. Thus PTAB is operating on a *pixel* basis rather than a *character* basis. This can come in handy when you are trying to get critical alignment between lines and text on the screen, or when you are trying to get a display to scroll smoothly in a horizontal direction (we shall see examples of this in upcoming chapters). Do not be confused between pixel row/columns and character row/columns. Pixels are accessed with the PSET, LINE, CIRCLE, and PTAB statements. Characters are spaced by larger rows and columns whose width and height depend on the size of the character being displayed.

FIGURE 4-18

Pixel tabbing with PTAB



```
1111111111
 1111111111
  1111111111
```

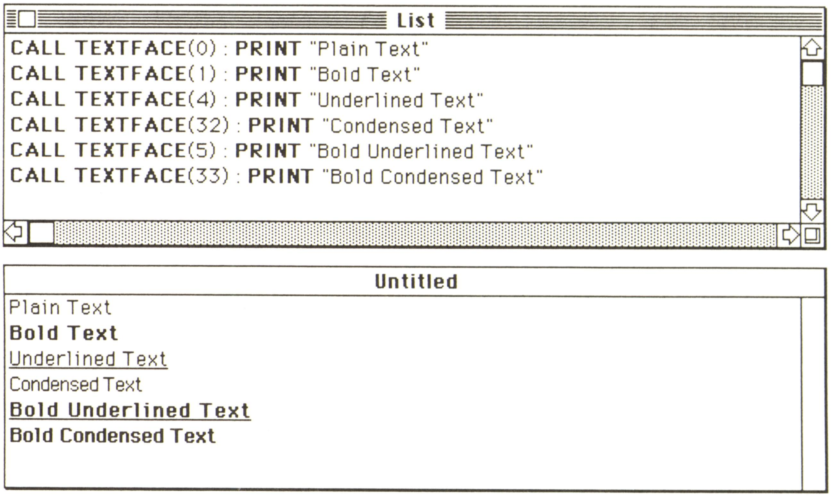
```
PRINT PTAB(2) "1111111111"
PRINT PTAB(4) "1111111111"
PRINT PTAB(6) "1111111111"
```

THE TEXTFACE STATEMENT

Before the Macintosh, computers could only print in one kind of text. Today, Microsoft BASIC allows us to print in several styles, or what it calls *textfaces.* A textface refers to the style of a letter or number, such as whether it is underlined, or bold, or italic. The Macintosh and Microsoft BASIC support several text styles, including plain, bold, underlined, outlined, shadowed, condensed (less space between characters), and extended. These styles are controlled with the CALL TEXTFACE statement. When CALL TEXTFACE is executed for a particular style, all subsequent PRINT strings will be printed in the selected style until a different CALL TEXTFACE statement is executed. To see how CALL TEXTFACE works, let's enter and run the program shown in Figure 4-19.

We have used CALL TEXTFACE in the program to change the style of the text before it is printed. In Figure 4-19, four different styles are combined to form six statements. The number inside the parentheses following CALL TEXTFACE selects the style.

The *plain text* face (0) is the standard face for all our Microsoft BASIC text. *Boldface* (1) is useful for titles, or printing messages that must stand out from other text. (Boldface makes each letter wider by a few pixels, so our total-column number of 61 doesn't hold when bold is used.) The *underlined* face (4) comes in handy for emphasizing text. Note that underline in BASIC underlines spaces as well as characters. *Condensed* (32) is similar to plain text, but the letters are closer together, as you can see in the example in Figure 4-19. Condensed text is useful whenever you are squeezed for room on the screen because it gives you more than the average 61 character-columns.



```
CALL TEXTFACE(0) : PRINT "Plain Text"
CALL TEXTFACE(1) : PRINT "Bold Text"
CALL TEXTFACE(4) : PRINT "Underlined Text"
CALL TEXTFACE(32) : PRINT "Condensed Text"
CALL TEXTFACE(5) : PRINT "Bold Underlined Text"
CALL TEXTFACE(33) : PRINT "Bold Condensed Text"
```

```
Plain Text
Bold Text
Underlined Text
Condensed Text
Bold Underlined Text
Bold Condensed Text
```

FIGURE 4-19

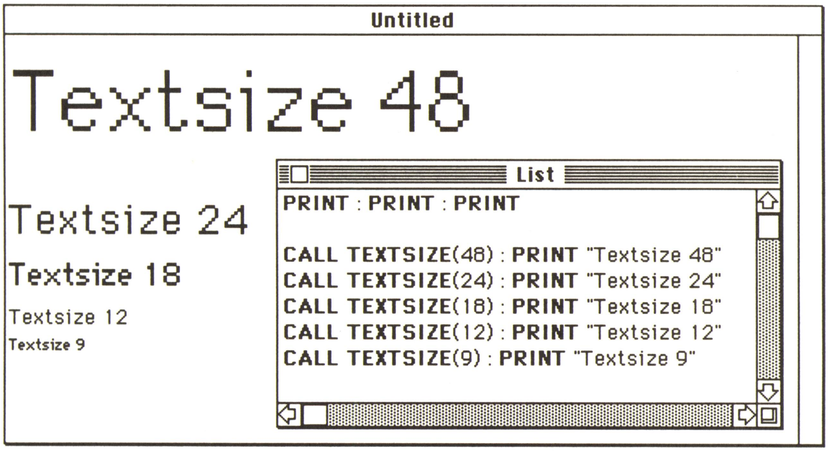
Printing different text styles using TEXTFACE

Combining textfaces

Textfaces can be combined. For example, you can have a string printed out as bold, underlined, and condensed all at the same time. This is done by giving the TEXTFACE statement the number *that is the sum of the individual styles you want.* For example, the numbers for the textfaces are 1 for bold and 4 for underlined. To get bold underlined text we would add the number for bold (1) to the number for underlined (4) and put their sum (5) in the CALL TEXTFACE statement, as we did in the second-to-last line in the program shown in Figure 4-19.

THE TEXTSIZE STATEMENT

As yet another reward for buying the Macintosh and Microsoft BASIC, you can change the *size* of the characters produced by your print statements. The size of text characters on the Macintosh screen is measured in *points.* There are 72 points in 1 inch, and so a 72-point character on the Mac screen is about 1-inch tall. Likewise, a 36-point character is about ½-inch tall, an 18-point character is about ¼-inch tall, and a 12-point character is about 3⁄16-inch tall. Microsoft BASIC uses the TEXTSIZE statement to control the size of the text that it prints. Like TEXTFACE, TEXTSIZE affects the output of subsequent PRINT statements, and remains in effect until the next TEXTSIZE statement is executed. To see how it works, let's enter and run the program shown in Figure 4-20.



```
Textsize 48

Textsize 24
Textsize 18
Textsize 12
Textsize 9
```

```
PRINT : PRINT : PRINT

CALL TEXTSIZE(48) : PRINT "Textsize 48"
CALL TEXTSIZE(24) : PRINT "Textsize 24"
CALL TEXTSIZE(18) : PRINT "Textsize 18"
CALL TEXTSIZE(12) : PRINT "Textsize 12"
CALL TEXTSIZE(9) : PRINT "Textsize 9"
```

FIGURE 4-20

Changing the size of the font using TEXTSIZE

In this program we print out text in six textsizes, from 48 point to 9 point. As you can see, when the letters get larger the individual dots they are made of appear as little squares. This may not be as pleasing as you would desire. What is happening is that BASIC is *scaling* the original 12-point Geneva font. This scaling involves a simple mathematical process that cannot make smooth letters. If you have the Font Mover program, you can move fonts that have been designed especially for the desired textsize. We will be covering the subject of fonts and font files in more detail in Chapter 15, QuickDraw.

Being able to use different textsizes allows you to create more effective titles and messages. Larger fonts are nice for banners, scrolling titles, people with poor eyesight, and children's programs. Larger fonts are also useful for alerting and grabbing attention. A smaller font, such as 9 point, is useful for labeling tiny graphs and charts. The only drawback is that above 14 points, most fonts (but not all, as we shall see) begin to show their jaggies.

PUTTING-IT-ALL-TOGETHER PROGRAM

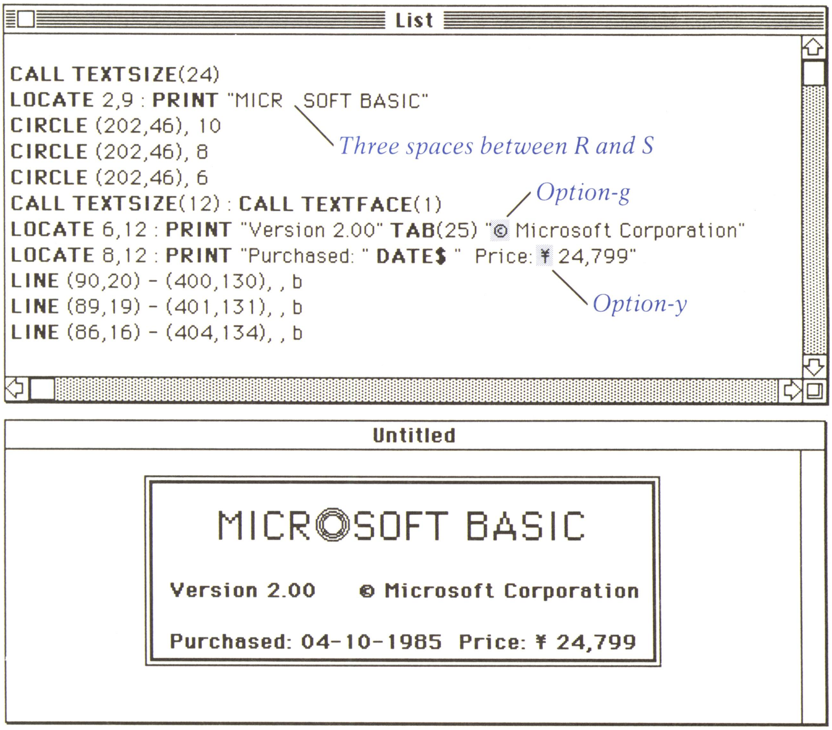
We have learned several new Microsoft BASIC statements for graphics and text work, including PSET, LINE, CIRCLE, LOCATE, PTAB, TEXTFACE, and TEXTSIZE. Let's put this knowledge to work for us.

A frequent function in any program is drawing labels, title boxes, banners, alert boxes, information boxes, and announcements. These objects often contain combinations of text and graphics. Before the Macintosh, most personal computers suffered when it came to mixing text and graphics—it was either impossible, or required special high-resolution equipment. Now you're going to see how easy it is on the Macintosh with Microsoft BASIC.

The program in Figure 4-21 produces the information box that tells the user this is Microsoft BASIC, its version number, the owner of the copyright, the purchase date, and the price in Japanese yen. The information is enclosed by the standard border for a typical Macintosh dialog box, like the one produced by the "About Microsoft BASIC" command in the Apple menu.

Study the program and see if you can understand how it works. You should be able to make sense of most of it. The following are some tips to help you see what it is doing:

1. The first line changes the textsize to 24 point and prints the words MICR   SOFT BASIC (that's right, the *O* is missing between the *R* and the *S* in MICROSOFT, and is replaced with 3 spaces).
2. Then we create the missing *O* with three CIRCLE statements.
3. Next, the textsize gets changed back to the normal 12 point, but we also switch the textface to bold. Now everything will get printed in bold.
4. Then we use LOCATE and PRINT statements to print the remaining information. The copyright symbol preceding *Microsoft Corp.* was obtained by pressing Option-g; the yen symbol was obtained by pressing Option-y. DATE$ was used to print the date.
5. Finally, we draw a border around the information. The inner box is made of two boxes that differ in size by one pixel. Then we move out two more pixels and draw a single-width box around the double-width box.



```
CALL TEXTSIZE(24)
LOCATE 2,9 : PRINT "MICR   SOFT BASIC"
CIRCLE (202,46), 10
CIRCLE (202,46), 8
CIRCLE (202,46), 6
CALL TEXTSIZE(12) : CALL TEXTFACE(1)
LOCATE 6,12 : PRINT "Version 2.00" TAB(25) "© Microsoft Corporation"
LOCATE 8,12 : PRINT "Purchased: " DATE$ "  Price: ¥ 24,799"
LINE (90,20) - (400,130), , b
LINE (89,19) - (401,131), , b
LINE (86,16) - (404,134), , b
```

Three spaces between R and S

Option-g

Option-y

```
MICR◎SOFT BASIC

Version 2.00      © Microsoft Corporation

Purchased: 04-10-1985  Price: ¥ 24,799
```

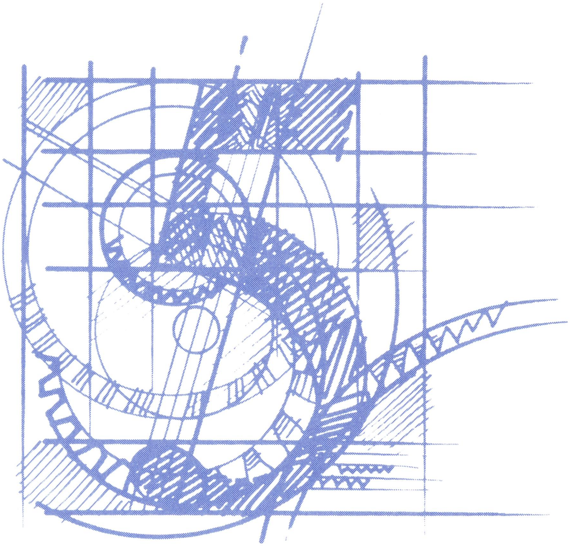
FIGURE 4-21

A summary program mixing text and graphics

REFRESH-YOURSELF WINDOW

This chapter has shown us how to do elementary graphics on the Macintosh screen. We have become familar with its coordinate system, both for pixels and for text. We know how to plot individual dots with the PSET statement, and how to draw lines, boxes, filled boxes, and circles anywhere on the screen. We know that the Macintosh "clips" images when we draw part of them outside the Output window. We can locate text anywhere on the screen, and change its size and style. We can print in columns, and we can control—down to the pixel—where our text starts with PTAB.

In the next chapter we will take a quantum leap in programming skill by learning how BASIC "remembers" numbers and letters. And we'll see how we can use Microsoft BASIC to manipulate the numbers and letters in our programs.



Variables and INPUT

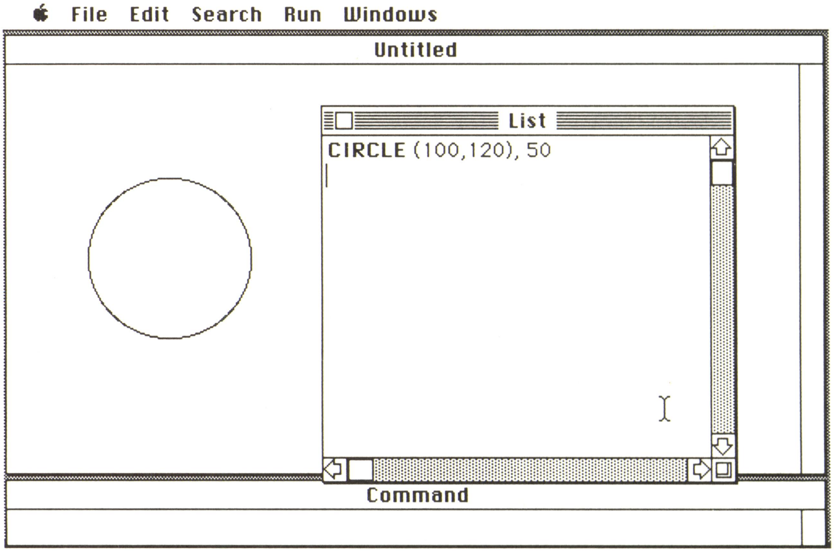
This chapter introduces one of the key concepts in programming: the variable. *One might say that variables are the soul of programming. Without variables a program is rigid and inflexible: It can perform very simple tasks, but (as we've seen in the examples so far in this book) it can't change its behavior to respond to what the program's user tells it to do. A program with* variables is more "alive"—it can ask you questions, and change its operation depending on what you tell it.

One of the simplest uses for variables is in handling information that the user of the program types in at the keyboard. For this reason, we are also going to introduce the INPUT statement in this chapter. In previous chapters you learned about PRINT and about a variety of graphics statements such as LINE and CIRCLE. These statements permit *the program* to send information out *to the user*, in the form of printing or drawings in the Output window. INPUT does the opposite: It permits *the user* to send information *to the program*. Taken together, variables and the INPUT statement are essential for creating powerful and exciting programs.

Because variables are such a powerful idea, they take a little getting used to. For this reason we're going to discuss them in stages, and look at them in several different ways.

THE SAME PROGRAM WITH AND WITHOUT VARIABLES

To introduce ourselves to variables we're going to create two short programs that do the same thing. The first doesn't use variables, while the second does. Figure 5-1 shows the first program, and the result in the Output window when it is executed. Enter the program line in your List window, then choose Start from the Run menu (or press Command-R).



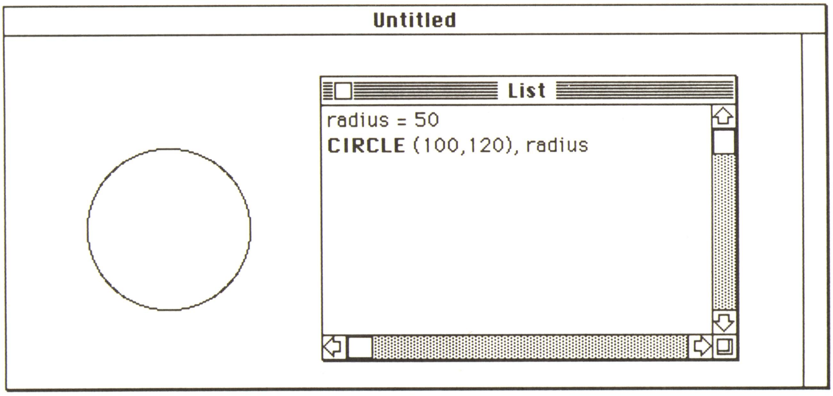
```
CIRCLE (100,120), 50
```

FIGURE 5-1

Drawing a circle without variables

FIGURE 5-2

Drawing a circle with a variable



```
radius = 50
CIRCLE (100,120), radius
```

This simple, one-line program puts a circle in the Output window. As explained in the last chapter, the numbers in parentheses are the coordinates of the center of the circle: The center is 100 pixels from the left edge of the Output window, and 120 pixels from the top edge. The circle's radius is given by the last number, 50. All this should be familiar to you.

Now we're going to write a program that generates the same circle, but does so in a different way. This new program is shown in Figure 5-2. Drag the pointer through the program line currently in the List window and press Backspace, then enter and run the new program lines.

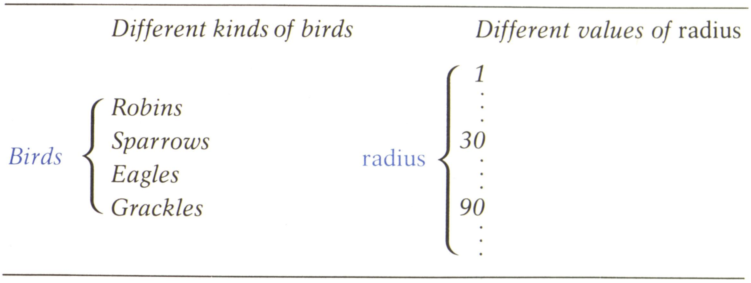
Notice the word *radius* in the program: It's an example of a *variable.* How does it work?

WHAT VARIABLES DO

Suppose there was only one kind of bird in the entire world—the robin, let's say. If this were true, there wouldn't be separate words for *birds* and *robins:* All robins would be known only as *robins* (or perhaps *birds*). But as soon as a second kind of bird appears on the scene—the sparrow, let's say—we would find that we *do* need separate words, first for the individual kinds of birds, *robins* and *sparrows,* and second for the *category* into which both robins and sparrows fall, namely *birds.*

FIGURE 5-3

The category of "birds" and the variable radius *can be given different values*



In a similar way, we now have a word in our program that describes a whole category or range of things: the variable *radius.* This variable is given a *value* of 50 by the line *radius* = *50.* This same line could also have given *radius* other values, such as 27 or 95. When we talk about the variable *radius* in the program, we're talking about a general category that can take on a specific value like 50 or 27, just as when we're talking about *birds,* we're talking about a general category that can have the specific value of *robins* or *sparrows* or *eagles* or *grackles.* This concept is illustrated in Figure 5-3.

THE ASSIGNMENT STATEMENT

The program line *radius* = *50* is called an *assignment statement,* because it assigns, or gives a value to, the variable *radius.* In a similar way, if someone said, "There's a bird," and you said, "What kind?" and they answered, "It's a grackle," then they would be assigning the specific value of *grackle* to the variable *bird.*

The assignment statement uses the *equal sign* to indicate that we are giving a specific value to the variable name. This is not the same way that an equal sign is used in arithmetic and mathematics. There, the equal sign means—as its name implies—that whatever is on the left side of the equal sign has the same value as whatever is on the right side. Thus in arithmetic you would say $5 = 3 + 2$.

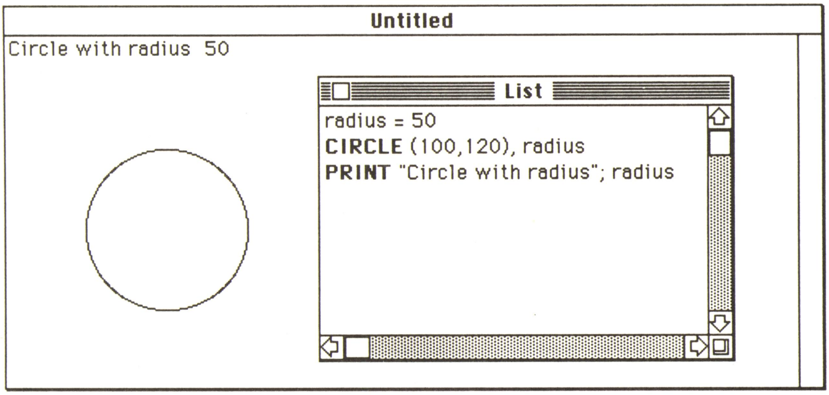
In BASIC, on the other hand, it doesn't make sense to put a number on the left side of the equal sign: We can't assign a value to a number because it already *is* a value. In a BASIC assignment statement, what goes left of the equal sign is always a *variable name,* and what goes right of the equal sign is either a specific *value,* like 25, or a variable which already has a specific value, as we'll see later.

MULTIPLYING OUR PROGRAM'S POWER WITH VARIABLES

What can variables do for us in a program? Let's see how we can use the variable we've created by adding a new line to our program, as shown below in Figure 5-4.

Notice that although we assigned *radius* the value of 50 only once, our program uses this value twice: once in the PRINT statement to print the caption with the correct value of the radius, and again in the CIRCLE statement to draw the circle with a radius of 50.

The ability to use the same variable for different purposes can be a real convenience. For example, suppose we want to change the program to draw a circle with a different radius. To do this we only need to change



```
Circle with radius  50
```

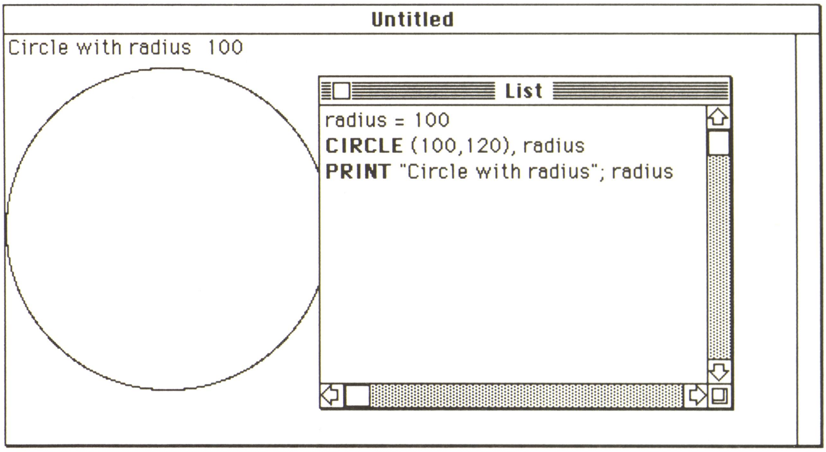
```
radius = 50
CIRCLE (100,120), radius
PRINT "Circle with radius"; radius
```

FIGURE 5-4

Adding a PRINT statement with a variable

FIGURE 5-5

The circle changed to a radius of 100



```
Circle with radius  100
```

```
radius = 100
CIRCLE (100,120), radius
PRINT "Circle with radius"; radius
```

the value of the variable. Let's backspace over the *50* in the *radius* variable and change it to *100,* as shown in Figure 5-5. When we run the program a new circle is drawn, as shown in the Output window of Figure 5-5, and the radius number printed in the caption also changes.

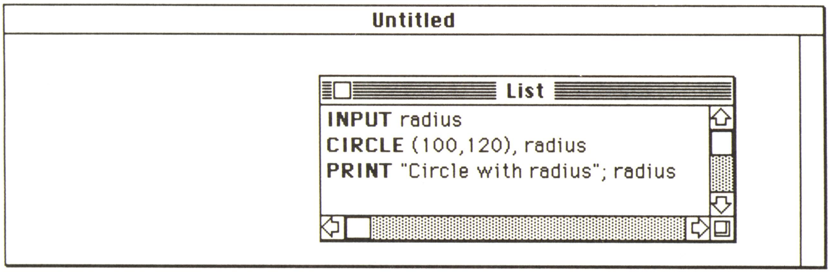
This is a good example of how making *one* change in a program can change *two* elements of the program's output. You could say that we have multiplied our programming power by two. As we progress, you'll see that variables can actually multiply our programming power thousands of times, but in the same sort of way: By changing one thing in a program, we can cause many changes in the program's operation.

THE INPUT STATEMENT

We mentioned earlier that the INPUT statement gives our programs the power to request information from the person using the program. Let's modify the circle program we've been working with to see how this works. We'll backspace over the radius line to erase it, then type the line shown in Figure 5-6 in its place.

FIGURE 5-6

The program with INPUT added



```
INPUT radius
CIRCLE (100,120), radius
PRINT "Circle with radius"; radius
```

We get a different result when we run this program than what we're used to: Instead of the List window disappearing and quickly reappearing, it stays hidden behind the Output window. Moreover, the File and Search menus remain inactive, and there's a question mark in the upper left corner of the Output window with the insertion point flashing next to it. This is shown in Figure 5-7. What's happening?

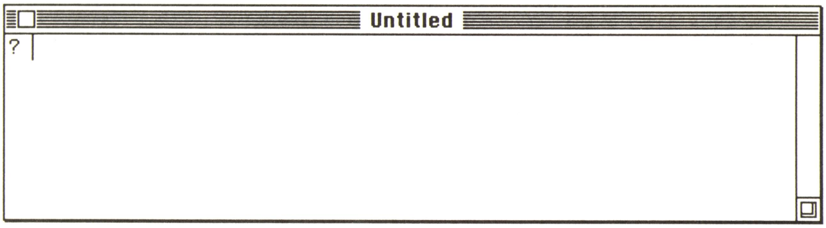
What is happening is that our program is still running. In fact, it's waiting for us to give it some information. When the INPUT statement is executed, the program prints a question mark and then waits for us to type a value for the variable *radius,* which follows the INPUT statement in the first line of the program.

Type: 20

As with most other input to the Macintosh, we must press the Return key after we have finished typing something in response to the INPUT

FIGURE 5-7

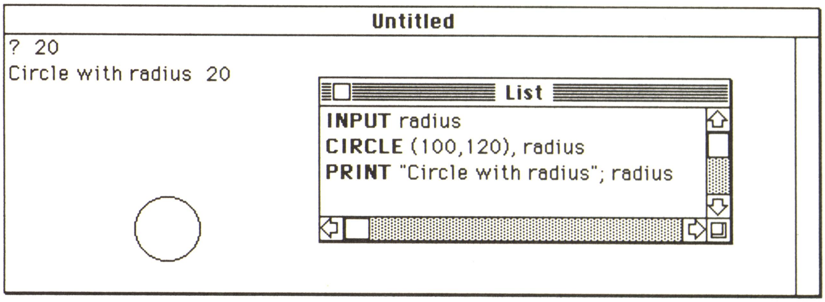
The Output window when INPUT is executed



```
?
```

FIGURE 5-8

The results of running the program with INPUT



```
? 20
Circle with radius  20
```

```
INPUT radius
CIRCLE (100,120), radius
PRINT "Circle with radius"; radius
```

question mark so that BASIC knows we're finished. The result is shown in Figure 5-8. The program draws the circle, prints the caption, and then ends the program and restores the List window. If you like, you can run the program several times, typing in different values for the radius.

Trouble with INPUT: *?Redo from start*

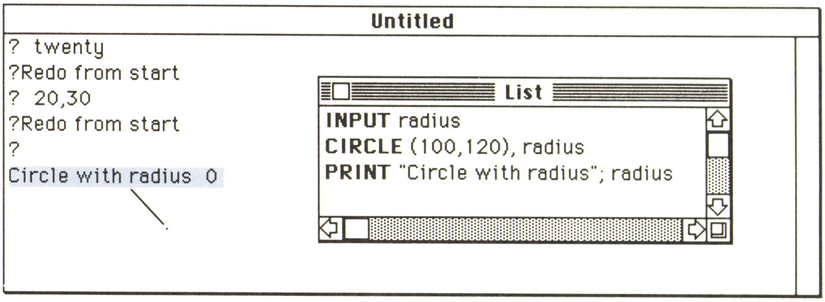
There is always the possibility that users will make mistakes when they type a response to the INPUT statement. INPUT is prepared to cope with at least some of these mistakes. When INPUT doesn't understand what you type, it prints the cryptic message, *?Redo from start.* This means that it wants you to start over again at the beginning, as indicated by a new question-mark prompt.

For instance, the user might type something that is not a number, or include some unexpected punctuation marks in the response. In either case INPUT will issue the *?Redo from start* message. Another possibility is that the user won't type anything, but will simply hit the Return key. In this case there will be no *?Redo from start* message because INPUT will simply assume that the value the user wants to INPUT is 0.

These three responses are demonstrated below in Figure 5-9. Notice that the circle on the Output window is just one pixel, which is a circle with a radius of 0.

FIGURE 5-9

Mistakes you can make with INPUT



```
? twenty
?Redo from start
? 20,30
?Redo from start
?
Circle with radius  0
```

```
INPUT radius
CIRCLE (100,120), radius
PRINT "Circle with radius"; radius
```

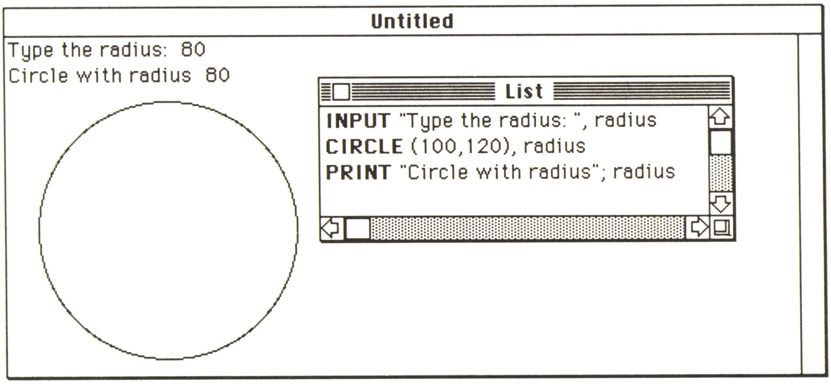
Adding a prompt to INPUT

The question mark, which the INPUT statement prints when it wants us to type something, is not very informative about what it wants us to type. However, INPUT has a feature that lets the program print a *prompt,* or message, instead of the question mark. Let's try it out by inserting the string shown in Figure 5-10 into our existing INPUT statement. (Don't forget the quotes, the space after the colon, and the comma at the end.)

Now whenever we run the program it prints the instruction we inserted in the INPUT statement instead of simply displaying a question mark. You can imagine how much easier it will be for users to run our program, now that it tells them exactly what it wants them to do instead of making them guess. Let's respond to the prompt by typing *80.* The result when we press Return is shown in Figure 5-10.

Commas and semicolons in prompts

Using a *comma* after the prompt, as we did in Figure 5-10, tells INPUT not to print the question mark as part of the prompt. But since the question mark is always followed by a space, we must add a space ourselves when we suppress the question mark by using a comma, as we did following *radius:* in our example. If you want the question mark and the prompt at the same time, you can use a *semicolon* in place of the comma, as we'll see in the next example.

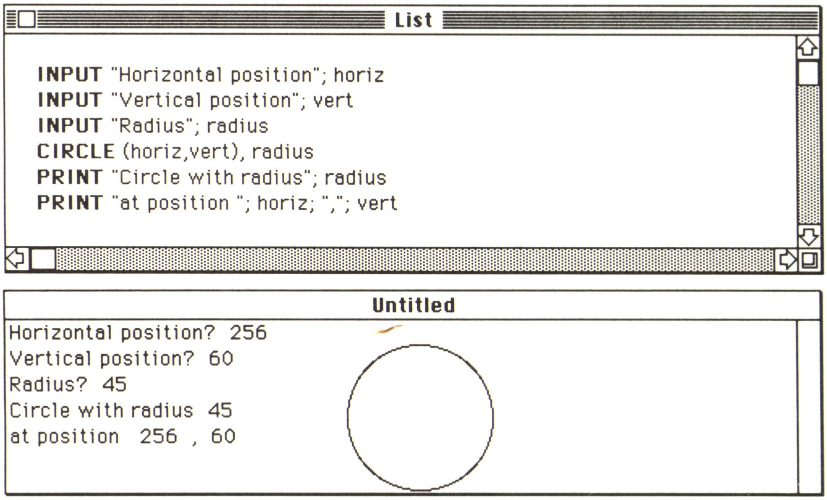


```
Type the radius: 80
Circle with radius 80
```

```
INPUT "Type the radius: ", radius
CIRCLE (100,120), radius
PRINT "Circle with radius"; radius
```

FIGURE 5-10

INPUT with a prompt



```
INPUT "Horizontal position"; horiz
INPUT "Vertical position"; vert
INPUT "Radius"; radius
CIRCLE (horiz,vert), radius
PRINT "Circle with radius"; radius
PRINT "at position "; horiz; ","; vert
```

```
Horizontal position? 256
Vertical position? 60
Radius? 45
Circle with radius  45
at position  256 , 60
```

FIGURE 5-11

The program with three variables

Multiple variables

Although we've used only one variable so far in our program, there is in fact no limit to the number a program can contain. Let's spruce up our example so we can draw circles wherever we want. The program shown in Figure 5-11 uses three variables instead of only one.

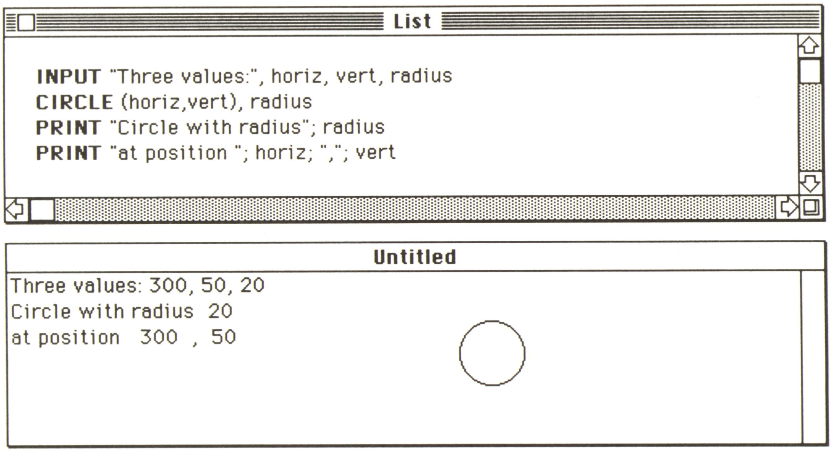
The new variables are *horiz* and *vert*, which of course stand for *horizontal position* and *vertical position*. They work in much the same way that *radius* does, except that they supply the horizontal and vertical coordinates to the CIRCLE statement. By changing our answers to the prompts, we can now change not only the size of the circle, but also its position. Of course, as you learned in the last chapter, if you make the horizontal and vertical coordinates too large you won't be able to see the circle: Part of it will be outside the Output window.

Multiple variables in one INPUT statement

Rather than having separate INPUT statements for each variable, we can include several variables in a single INPUT statement, although we will still be limited to one prompt. Let's rewrite our program as shown in Figure 5-12 on the next page.

FIGURE 5-12

Multiple variables in INPUT



```
INPUT "Three values:", horiz, vert, radius
CIRCLE (horiz,vert), radius
PRINT "Circle with radius"; radius
PRINT "at position "; horiz; ","; vert
```

```
Three values: 300, 50, 20
Circle with radius  20
at position  300  ,  50
```

Although this is not a big improvement in the present program, the principle can be useful in situations when we're using INPUT to capture a list of similar items, such as the test scores in a teacher's class. Notice that the three variables are separated by commas in the INPUT statement, and that when the user types in the three values, they must also be separated by corresponding commas.

VARIABLE NAMES

How do we decide what to call our variables? Could we have also used names like *horizontal*, or *horizontal.position?* Yes: Both of these names are valid. However, there are some names we can't use. Here are the rules for naming a variable:

- It must have no more than 40 characters.
- It can use any letters, and the digits 0 to 9.
- The only punctuation mark allowed is the period.
- The first character must be a letter.
- No spaces may be used (this is different from the way programs are named, so be careful).
- No reserved words may be used (you can tell if you've used one, because it will be printed in boldface when you press Return).
- Upper- or lowercase can be used in any combination, but BASIC can't tell the difference between cases, so it will consider *RADIUS*, *radius*, and *Radius* to be the same variable.

Little variable boxes

One way to think about variables is to imagine little boxes located somewhere inside the Macintosh. Each box has a name written on it, and contains a slip of paper with a number written on it. (Later we'll see that the pieces of paper can have character strings written on them as well.) Figure 5-13 illustrates this concept.

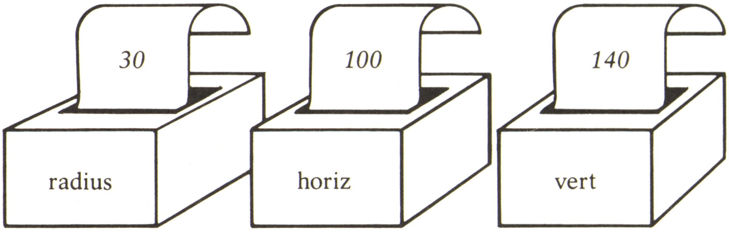
Of course, these boxes don't really exist inside the computer, but their electronic equivalents exist in the memory chips within the Macintosh. However, the box idea is a useful way to think about variables. Each of these boxes represents a variable. The name of the box corresponds to the name of the variable, and the number on the paper corresponds to the value the variable has at any particular moment.

You can think of the "boxes," or variable names, as permanent parts of the program: The programmer creates them, and they do not change unless the programmer changes them. The "pieces of paper," or variable values, on the other hand, are more temporary. They are either fed to the program with an INPUT statement by the person using the program, or they are created by the program itself, perhaps using an assignment statement like *radius* = 50.

In the next section, we'll use variables in a somewhat different way: to perform arithmetic calculations. This will show you more about how variables are used and the role they can play in programming.

FIGURE 5-13

Little variable "boxes"



ARITHMETIC WITH VARIABLES

So far we've worked with variables that are given values by the user and then used to generate circles of different sizes in different locations on the Output window. To extend our familiarity with variables, let's see how they can be used to solve arithmetic problems.

Our goal in this section is to write a program that calculates how much it will cost us to buy a certain number of items at a fixed price per item. For instance, if we bought 12 oranges at 20 cents per orange, the total would be 12 times $0.20, or $2.40.

To make the problem a little more interesting, let's calculate the sales tax and add it to the result. We'll assume that the tax rate is always 10 percent. To calculate the tax, we first find the subtotal, which is $2.40 in the above example, and multiply it by 0.10, which is 10 percent expressed as a number (since BASIC doesn't understand the percent sign). So the tax would be 0.10 times $2.40, which is $0.24. The total would be the original subtotal plus the tax: $2.40 plus $0.24, or $2.64.

Obviously, this total is correct only for this number of items at this cost per item. But what if we want to find the total costs for *various* items, such as oranges, apples, and bananas, at *various* prices? Of course: We'd create *variables* whose values could change with the item being input. So we would have to design our BASIC program so that it would perform the correct calculations no matter what different values we input to the program. In this case, the general structure of the problem is:

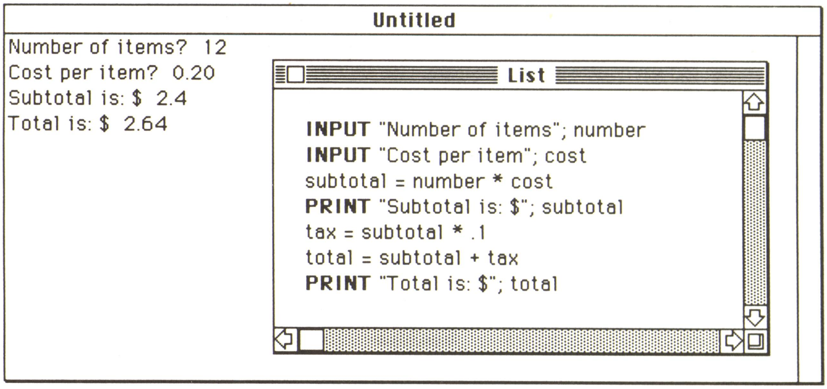
```
subtotal = number * cost
tax = subtotal * taxRate
total = subtotal + tax
```

(You will notice that we've used the asterisk to indicate multiplication, as BASIC requires.)

Let's use these variables in a program that asks us how many items we're buying and how much they cost, and then performs the above calculations for us. The program is shown in Figure 5-14 on the following page, together with the results when we input the numbers from our oranges example in response to the prompts. (Be careful to enter the cost as a decimal fraction, so that the value will be expressed in dollars.)

FIGURE 5-14

A program to find the total cost of a number of items



```
Number of items?  12
Cost per item?  0.20
Subtotal is: $  2.4
Total is: $  2.64
```

```
INPUT "Number of items"; number
INPUT "Cost per item"; cost
subtotal = number * cost
PRINT "Subtotal is: $"; subtotal
tax = subtotal * .1
total = subtotal + tax
PRINT "Total is: $"; total
```

Values and variables

The important thing to notice about this program is how the variables are used. In the third line we say *subtotal = number * cost,* but earlier we said that the equal sign in BASIC means that we're assigning the value on the right side of the sign to the variable on the left. How can we assign a value, when we don't see a number on the right?

The answer is that the variables *number* and *cost* are given values by the INPUT statements in the first two lines of the program. BASIC then takes these values, multiplies them together, then assigns the resulting value to the variable *subtotal.* The moral is that what is on the right side of an assignment statement doesn't need to be a specific number (like 2.40); it can also be an expression with variable names and operators (like *), provided that BASIC can figure out what the value of this expression is.

This technique—expressing the value of a variable in terms of other variables—is a very powerful idea, and one that lies at the heart of the programming process.

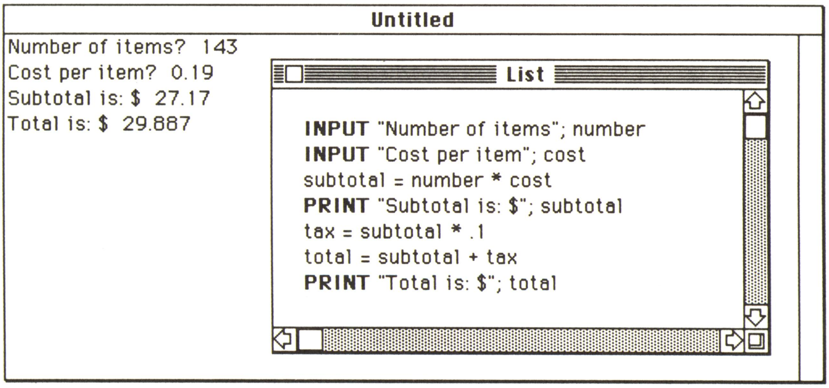
Rounding off

We've cheated somewhat in this example by using numbers that don't generate fractions of a cent. Suppose, however, we had 143 items which cost $0.19 each. Then the results would look as shown in Figure 5-15.

If we were doing the calculation ourselves, we'd *round off* $29.887 to $29.89. But apparently BASIC isn't smart enough to know that we're talking about money and that printing fractions of a cent isn't appropriate. In future chapters we'll show you several ways to get BASIC to round off numbers and avoid this kind of problem.

FIGURE 5-15

Variables that result in fractions of a cent



```
Number of items?  143
Cost per item?  0.19
Subtotal is: $  27.17
Total is: $  29.887
```

```
INPUT "Number of items"; number
INPUT "Cost per item"; cost
subtotal = number * cost
PRINT "Subtotal is: $"; subtotal
tax = subtotal * .1
total = subtotal + tax
PRINT "Total is: $"; total
```

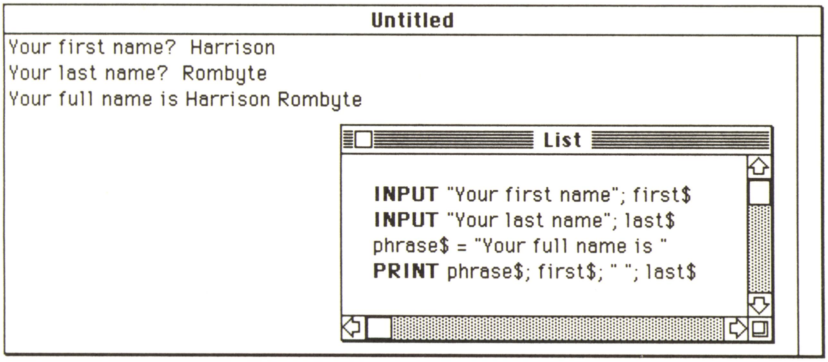
STRING VARIABLES

So far we have only talked about variables that are assigned *numerical* values. However, it is also possible for variables to have *strings* as values. Strings, as we learned in Chapter 2, are groups of letters, numbers, and punctuation. Sentences, words, and paragraphs are all examples of strings.

First let's move and resize the List window as shown below in Figure 5-16, then enter and run the program listed there, which uses strings as variables.

FIGURE 5-16

A short program using string variables



```
Your first name?  Harrison
Your last name?  Rombyte
Your full name is Harrison Rombyte
```

```
INPUT "Your first name"; first$
INPUT "Your last name"; last$
phrase$ = "Your full name is "
PRINT phrase$; first$; " "; last$
```

Strings versus numbers

One important difference between string variables and the numerical variables we've been discussing up to now is that *string variables always end with a dollar sign.* The dollar sign tells BASIC that we're not dealing with a number, but with a string.

Remember when we typed the string *twenty* instead of the number 20 into our circle-drawing program, and INPUT gave us its *?Redo from start* error message? It's this sort of problem we prevent by using the dollar sign. Once BASIC sees the dollar sign and knows that we are looking for a string variable, it allows us to input a string without hassling us with an error message.

The reason that BASIC needs to know whether a variable is a string or a number is that it stores these variables differently. To recall our analogy of the little boxes, it uses different *kinds* of boxes to store strings than it does to store numbers. If we try to put a number into a box intended for a string, or vice versa, it won't fit.

String constants

Notice the program line *phrase$ = "Your full name is "* in our last example. The phrase in quotes is an example of a *string constant.* In one of our previous programs the line *radius = 25* assigned a constant value of 25 to the variable *radius.* Here, *"Your full name is "* is a constant, which the statement assigns to the variable *phrase$.*

Longer strings and the List window

Let's write a somewhat longer program that uses string variables. Because the lines in this program are so long, we need to use a different technique in handling our windows:

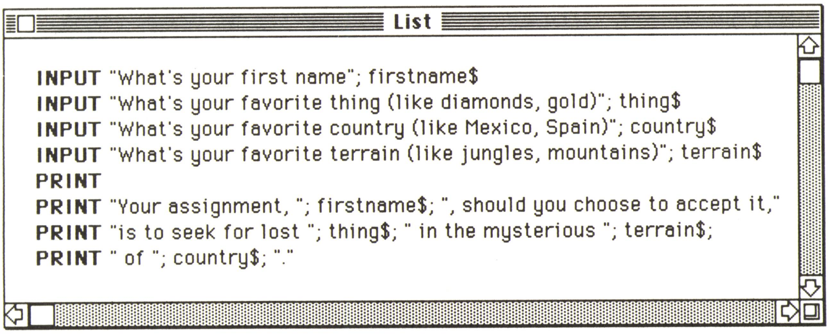
Dbl-click: on the List window's title bar

This expands the List window to the full size of the screen, as shown in Figure 5-17 on the next page.

Type: the listing shown in Figure 5-17

FIGURE 5-17

Listing of the Adventure program



```
INPUT "What's your first name"; firstname$
INPUT "What's your favorite thing (like diamonds, gold)"; thing$
INPUT "What's your favorite country (like Mexico, Spain)"; country$
INPUT "What's your favorite terrain (like jungles, mountains)"; terrain$
PRINT
PRINT "Your assignment, "; firstname$; ", should you choose to accept it,"
PRINT "is to seek for lost "; thing$; " in the mysterious "; terrain$;
PRINT " of "; country$; "."
```

Now that we've done all this typing to create our program, let's save it with a descriptive name, in case we ever want to return to it and make it more "adventuresome."

Choose:	Save As... from the File menu
Type:	Adventure
Click:	the Save button

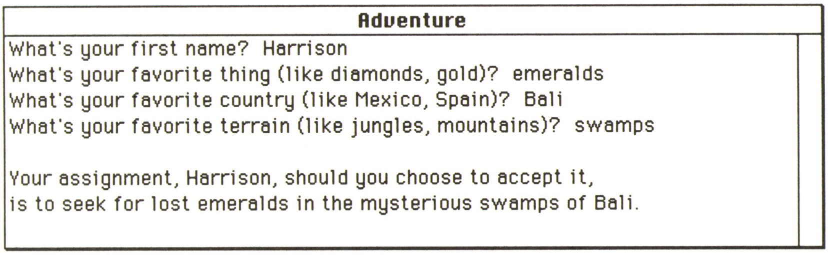
We must also use a new technique to run this program. Since the List window now takes up the entire screen, we need to get rid of it before we run the program. Otherwise, as soon as the program is finished, the List window will reappear and blot out the Output window. When this happens, the output that the program placed in the Output window is gone.

Click:	the List window's close box
Choose:	Start from the Run menu

When the program executes, the Output window remains on the screen and we can type our responses to the prompts, as shown in Figure 5-18.

FIGURE 5-18

Output of the Adventure program



```
What's your first name? Harrison
What's your favorite thing (like diamonds, gold)? emeralds
What's your favorite country (like Mexico, Spain)? Bali
What's your favorite terrain (like jungles, mountains)? swamps

Your assignment, Harrison, should you choose to accept it,
is to seek for lost emeralds in the mysterious swamps of Bali.
```

THE REM AND END STATEMENTS

Before we leave this chapter, we're going to introduce two statements that may seem somewhat superfluous. However, both statements will prove to be very useful in the longer and more complicated programs we will be encountering in chapters to come.

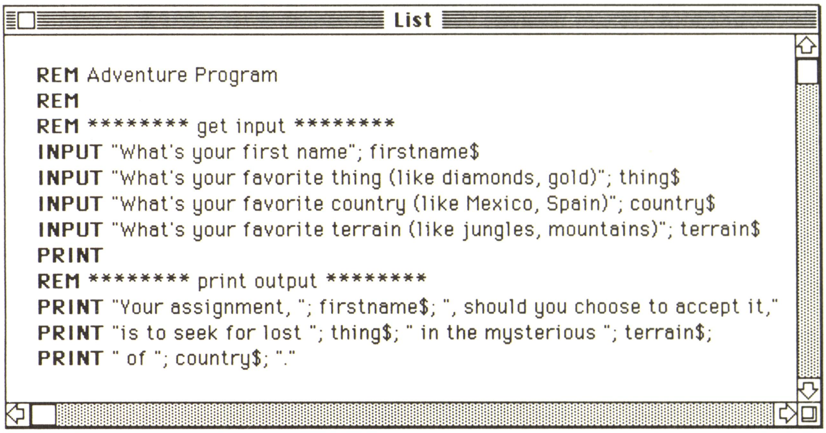
Annotated listings: The REM statement

In some ways it's easier to write a program than it is to understand what a program that's already been written does. When you're writing a program yourself, you know what each program line is for. But when you look at a program someone else has written, or even when you look at a program you wrote yourself some time ago, you'll find that it can be quite difficult to figure out what's happening. As your programs grow longer and more complicated, this problem will become more severe.

It would be nice if there were a way to include notes and comments in our program listings to remind ourselves (or someone else) what we intended a particular section of the program, or a particular statement, to do. There is such a way: the REM statement. REM is short for *Remark:* In effect it tells BASIC, "Ignore everything else on this line, it's just a comment to the programmer."

REM can be used to provide a title for the program, so you won't forget what the purpose of the program is, and to title different sections of the program or individual lines.

Figure 5-19 shows our Adventure program with some REM statements added. Because REM statements are comments to the programmer rather than statements that BASIC needs to understand, they can be written in any format (provided they start with REM). Thus styles vary greatly among programmers in the use of REM statements. However, there is one



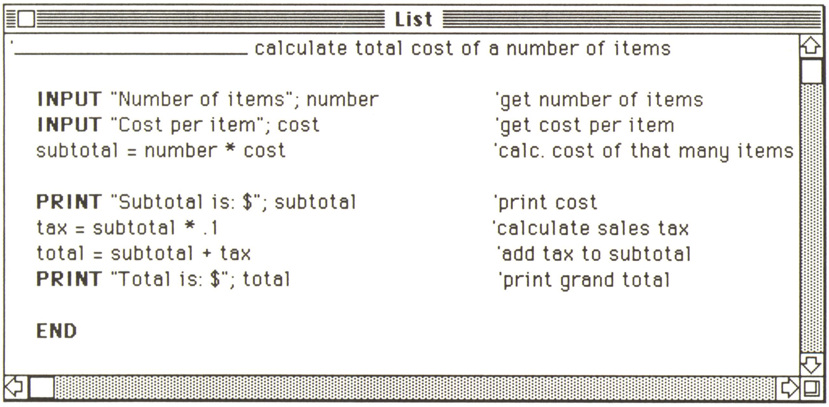
```
REM Adventure Program
REM
REM ******** get input ********
INPUT "What's your first name"; firstname$
INPUT "What's your favorite thing (like diamonds, gold)"; thing$
INPUT "What's your favorite country (like Mexico, Spain)"; country$
INPUT "What's your favorite terrain (like jungles, mountains)"; terrain$
PRINT
REM ******** print output ********
PRINT "Your assignment, "; firstname$; ", should you choose to accept it,"
PRINT "is to seek for lost "; thing$; " in the mysterious "; terrain$;
PRINT " of "; country$; "."
```

FIGURE 5-19

The Adventure program with REM statements

FIGURE 5-20

A program with single-quote comments and END statement



```
'______________________ calculate total cost of a number of items

  INPUT "Number of items"; number           'get number of items
  INPUT "Cost per item"; cost               'get cost per item
  subtotal = number * cost                  'calc. cost of that many items

  PRINT "Subtotal is: $"; subtotal          'print cost
  tax = subtotal * .1                       'calculate sales tax
  total = subtotal + tax                    'add tax to subtotal
  PRINT "Total is: $"; total                'print grand total

  END
```

rule that experienced programmers agree on: It's better to have too many remarks in your program rather than too few. That way, when you go back to look at a long program listing you won't find yourself uttering the well-known programmer's lament, "Oh, why didn't I use more remarks?"

Shorthand form of REM: The single quote

You may find that all the REM statements make your listing look unnecessarily cluttered. Or, you may want to add a remark to an existing program line, rather than making each remark a separate line. In either case, the single-quote mark (') serves the purpose. Figure 5-20 shows a program from earlier in this chapter, rewritten to include a title, and an explanatory remark on every line, using single-quote marks.

The END statement

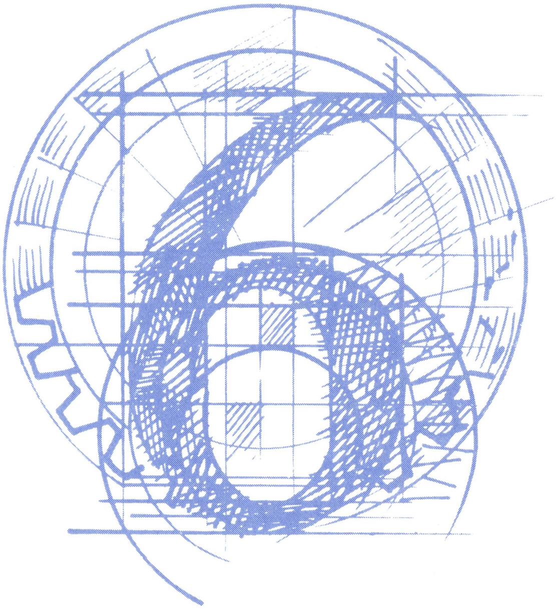
Notice in Figure 5-20 that we've also included the reserved word END. Why was that? Well, everything must come to an end, including this chapter and each of the programs we write. BASIC's END statement marks the end of a program.

What does the END statement do? In the short programs we've written so far, we haven't really needed it: BASIC went through our programs step by step, and when it finished all the steps, it stopped. However, there will be cases, especially when we explore subroutines, when we will want the program to end before it reaches the last program line. In that case, END is a necessary statement. In the meantime, we'll use END because it makes it clear that the program is indeed over, and not that the last page of the listing has been lost.

REFRESH-YOURSELF WINDOW

We've discussed an important topic in this chapter: variables. We've explained what they are, and how to use them in a variety of situations. They are indispensable to programming, so you'll be seeing and learning more about them in future chapters. Among other things, you'll learn that there are more than the number and string variables we discussed in this chapter, and you'll find out how to change one kind of variable into another.

We also discussed the INPUT statement, which lets the user provide information that the program can use. We finished up with the REM and END statements which, while they may not be as exciting as variables or as useful as INPUT, nevertheless have their places in the rich hierarchy of BASIC commands, like foot soldiers in the service of the king.



Loops

If we can call variables the "soul" of BASIC, as we did in the last chapter, then perhaps we can call loops *the "muscle." Loops give our program the strength to do something over and over again. This idea of repetition is an important one in many aspects of our modern world. An assembly line that made only one tin can, and then had to be started over, would hardly be worth* the investment. But an assembly line that can turn out hundreds of cans a minute without human intervention is a valuable device.

So far in this book our programs have done what they do only once. Then they end, and must be restarted. Although this is appropriate for a few simple programs, most programming problems require that parts of the program be executed over and over again. This chapter will show you how to add the power of repetition to your programming skills.

We'll start out by describing the FOR...NEXT loop, a simple yet powerful programming structure that lets our programs do something a fixed number of times. Then we'll explain the WHILE...WEND loop, a somewhat more sophisticated looping structure that permits programs to do something over and over until a certain condition is met. And to help you understand the concept of WHILE...WEND loops, we'll introduce some *relational operators* such as "greater than" and "less than." Also, we'll show you how your program can interact with the mouse: finding out if the mouse button is pressed, and where the mouse pointer is located in the Output window. We'll close the chapter by telling you about the black sheep of BASIC statements, GOTO, and about another important programming concept called *labels.*

THE FOR...NEXT LOOP

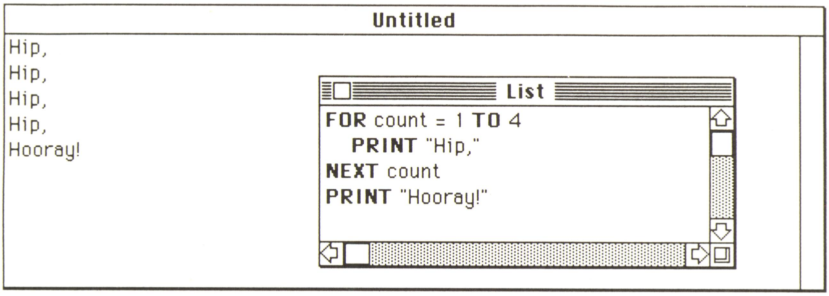
The FOR...NEXT loop is used whenever we want a program to do something a certain number of times, and we know how many times this is *before* the program enters the loop. It consists of a pair of BASIC statements, FOR and NEXT. These two statements are always used together to form a loop. If we try to use one without the other, we will get an error message.

The FOR statement comes at the beginning of the loop, and tells BASIC how many times we want to go through the loop. The NEXT statement comes at the end of the loop. It's responsible for figuring out if the loop has been executed the number of times specified by the FOR statement. If it hasn't, the NEXT statement causes the program to jump back to the FOR statement for another execution of the loop.

Figure 6-1 shows a simple example of a FOR...NEXT loop. It prints the word *Hip* four times, then prints the word *Hooray.* Although we might not write a computer program to do this in real life, it will serve as a demonstration of the principles involved.

FIGURE 6-1

An example of a FOR...NEXT loop



```
Hip,
Hip,
Hip,
Hip,
Hooray!
```

```
FOR count = 1 TO 4
  PRINT "Hip,"
NEXT count
PRINT "Hooray!"
```

Type in this program and run it. You'll notice that there's only one *PRINT "Hip,"* statement, yet the word *Hip* is printed four times. How can this be? The answer lies in the FOR and NEXT statements that surround the *PRINT "Hip,"* statement. Let's take the program apart piece by piece and see how it works.

The FOR statement

The FOR statement sets up the loop, and also tells BASIC where the beginning of the loop is. To set up the loop, FOR uses a variable called a *counter.* In our program, we've given this counter variable the name *count.* This variable, as its name implies, counts how many times the loop has been executed. It starts at the number just after the equal sign—1 in our program—and ends at the number after the TO—4 in our program. The first time through the loop *count* is 1, the second time it is 2, the third time 3, and the fourth and last time it has the value of 4. It increments—or adds 1 to itself—automatically each time through the loop. As a programmer you don't need to worry about doing this. It's as if somewhere out of sight BASIC were executing the statement *count = count + 1* each time the loop was executed.

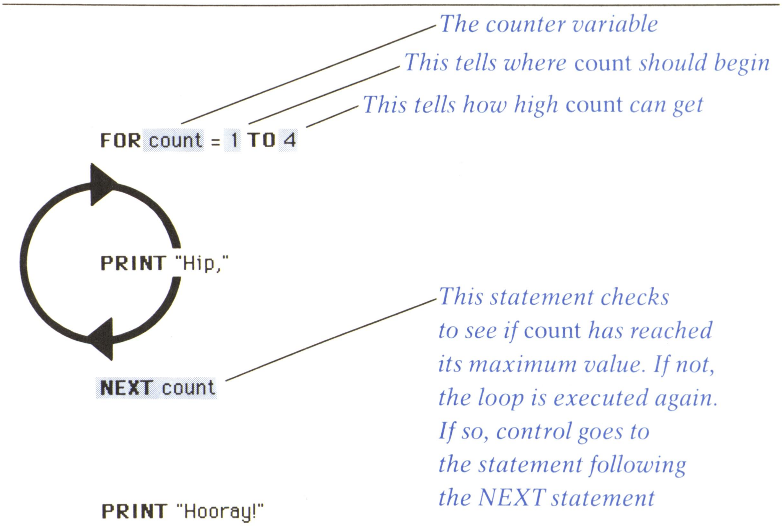
The NEXT statement

The NEXT statement has four functions: It tells BASIC where the end of the loop is, it figures out if the loop has been executed the specified number of times, and if not, it causes BASIC to go back to the beginning of the loop and do it again, and it increments the counter variable.

In our program, the first time BASIC gets to the NEXT statement it says, in effect, "*count* isn't equal to 4 yet; it's only equal to 1. So I'll increment it to 2, and then I'll go back to the statement just after FOR." The second and third times are similar: *count* is incremented, first to 3, and

FIGURE 6-2

The structure of the FOR... NEXT loop



```
FOR count = 1 TO 4
    PRINT "Hip,"
NEXT count
PRINT "Hooray!"
```

The counter variable

This tells where count should begin

This tells how high count can get

This statement checks to see if count has reached its maximum value. If not, the loop is executed again. If so, control goes to the statement following the NEXT statement

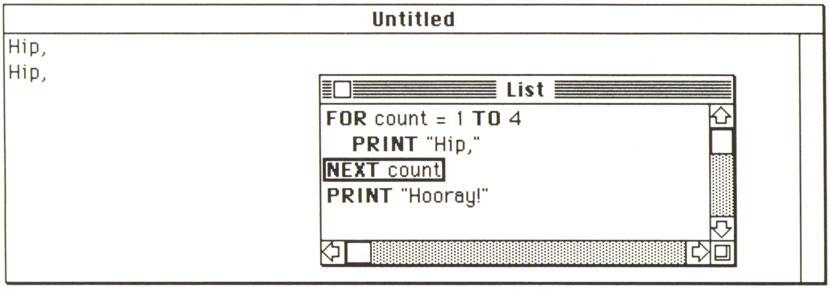
then to 4. The fourth time, however, BASIC gets to the NEXT statement and says, "Now *count* is 4, and that's as big as it's supposed to get, since 4 is the number after TO. So we'll stop looping now, and go on to the statement following NEXT." Which in our program is *PRINT "Hooray!"*. Figure 6-2 shows the major elements of the FOR...NEXT loop.

Indenting the body of the loop

Notice that the statement that is actually executed inside the loop, *PRINT "Hip,"* is *indented* to the right of the other program statements. This isn't necessary as far as BASIC itself is concerned, but it makes it much clearer to someone trying to understand the program listing what's inside a loop and what isn't.

The indentation is easily achieved with the Tab key, which indents the current line four spaces each time it's pressed. Usually, we'll have more than one line inside our loop, so BASIC cleverly continues to indent each line after we press the Return key. To go back out to the left side of the window—to the *NEXT count* statement in our case—we'll need to press the Backspace key four times.

It's a good idea to follow this style of indenting when writing any loop. In short programs such as this it doesn't really make much difference in clarity, but in long programs with complex loops you'll find it almost essential to keep track of what the program is doing.



```
Hip,
Hip,
```

```
FOR count = 1 TO 4
   PRINT "Hip,"
NEXT count
PRINT "Hooray!"
```

FIGURE 6-3

Stepping through the program

Stepping and tracing through the loop

A good way to get a feeling for how the FOR...NEXT loop works is to *step* through the program. You do this by choosing Step from the Run menu, instead of Start. Actually, it's faster to use the Command-T key combination: Hold down the Command key, and press T. A black box outlines the statement being executed, as shown in Figure 6-3. Each time you press Command-T, another statement is executed.

The FOR statement is executed first, then *PRINT "Hip,"* and then NEXT. Then control goes back to *PRINT "Hip,"* again, then to NEXT, and so forth until *Hip* has been printed four times. After the final *Hip,* the *PRINT "Hooray!"* statement is executed, and the program ends.

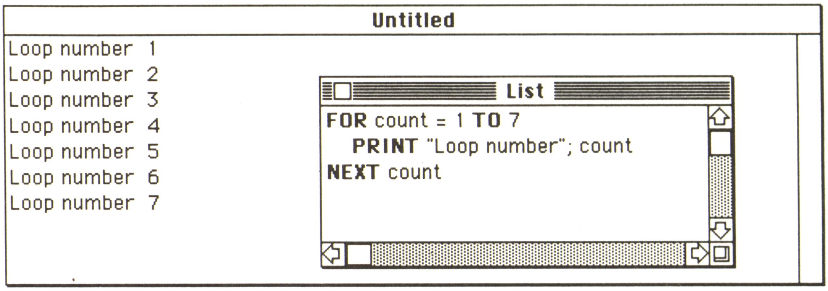
You might also try choosing Trace On from the Run menu, and then run the program. The effect is like Step, except that instead of having to press Command-T to draw a new box, the box moves automatically from line to line. If the List window disappears when you run the program, you can choose Show List *while the program's running* to bring the List window to the front where you can watch the box as each line is executed.

Using the counter variable

The counter variable is like any other variable in that your program can use its value if it wants. Let's see what this means by looking at the program in Figure 6-4 on the following page.

This program is a FOR...NEXT loop that counts from 1 to 7. Inside the loop, we print out the current value of the counter variable. Each time the loop is executed the counter variable is incremented by 1, until 7 is reached; at this point the program ends.

It's often useful for a program to use the counter variable inside the loop for more complex activities than simply printing it out. We'll show



```
Loop number  1
Loop number  2
Loop number  3
Loop number  4
Loop number  5
Loop number  6
Loop number  7
```

```
FOR count = 1 TO 7
   PRINT "Loop number"; count
NEXT count
```

FIGURE 6-4

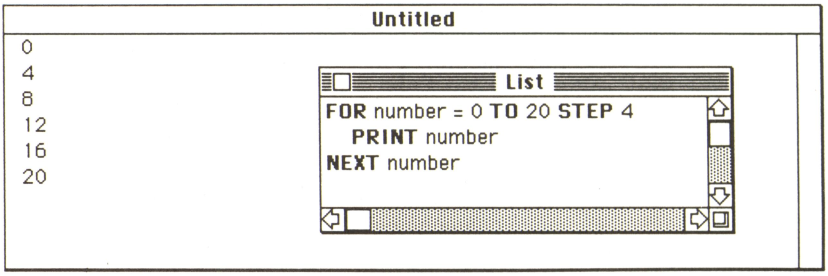
Printing the counter variable

examples of this later. However, it is very important that our programs *not change* the value of the counter. For instance, the statement *count = 3* inside a loop that used *count* as its counter variable would cause BASIC to lose track of where it was in the count, and in fact would produce a loop that would never end. Our programs can use the counter variable, but they shouldn't modify it.

The STEP option

So far we've implied that the counter variable must always be incremented by 1 each time through the loop, and this is in fact what happens unless we tell BASIC otherwise. However, by adding a word and a number to our FOR statement we can cause the counter to be incremented by whatever amount we want. The word to be added is STEP. Figure 6-5 shows a program in which the counter variable (which is called *number* this time) is incremented by 4 instead of 1.

Notice how, in this program, the counter variable number never takes on the intermediate values 1, 2, or 3. It starts out at 0, and the next time through the loop it's 4. The next time through it's 8, and so on, until the final value of 20 is reached.

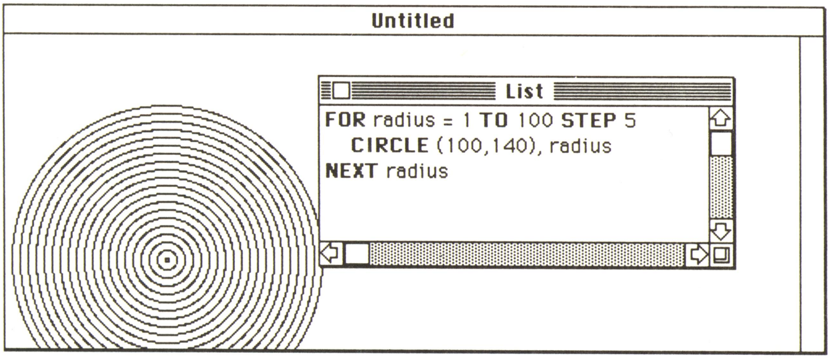


```
 0
 4
 8
 12
 16
 20
```

```
FOR number = 0 TO 20 STEP 4
   PRINT number
NEXT number
```

FIGURE 6-5

Using STEP to increment a counter variable



```
FOR radius = 1 TO 100 STEP 5
   CIRCLE (100,140), radius
NEXT radius
```

FIGURE 6-6

Drawing concentric circles with STEP

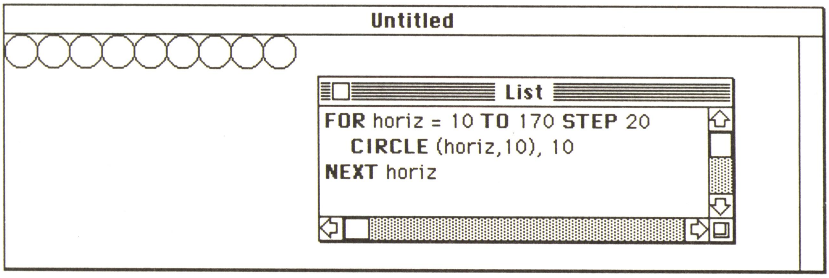
Figure 6-6 shows another example of a FOR...NEXT loop using STEP. It draws a series of concentric circles. In this case, however, we need to use a STEP greater than 1, because if the counter variable *radius* were incremented by 1 each time, the circles would be so close together that the result would be a solid black disk. So we use a STEP of 5 to separate each circle from the next by 5 pixels.

You will also notice in this program that we have used the counter variable *radius* to specify the radius of the circle as well as the count. It's serving double duty: first to count how many times to go through the loop, and second as a variable in the CIRCLE statement to specify the radii of our circles.

Nested loops

It is possible to put one loop inside another. In fact, this is a common situation in programming. Loops within loops are called *nested loops.* We're going to introduce nested loops in a series of three short programs that draw circles on the screen.

The first program doesn't really have any nested loops but is a component of the other two programs. It's shown in Figure 6-7.



```
FOR horiz = 10 TO 170 STEP 20
   CIRCLE (horiz,10), 10
NEXT horiz
```

FIGURE 6-7

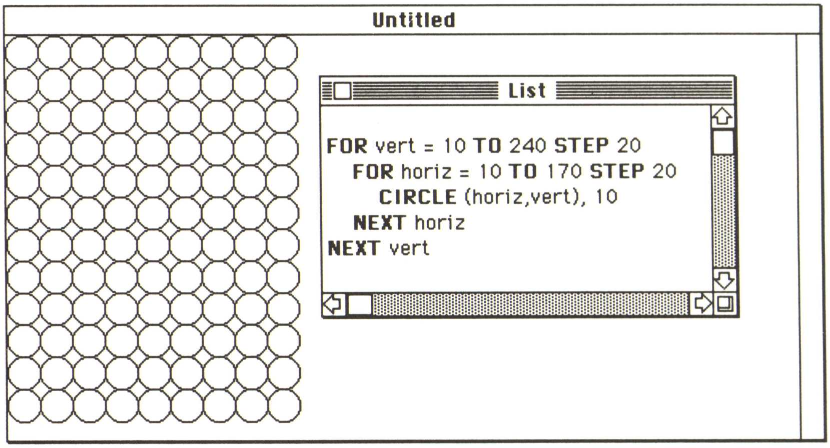
Drawing a row of circles

This program draws a row of circles across the top of the screen. It does this by using the counter variable *horiz* to change the horizontal coordinate of the center of the circle in the CIRCLE statement each time a circle is drawn. We've chosen the starting and ending points of the counter variable, 10 and 170, so that we can fit nine circles in the space to the left of the List window. The step size, 20, is the same as the diameter (twice the radius) of the circles, so they will just touch each other when they're all lined up in a row.

The program shown in Figure 6-8 actually contains a nested loop. It takes our first program, which drew a horizontal line of circles, and puts it inside another loop. The purpose of this new outer loop is to change the *vertical* coordinate of the circle that's being drawn. The effect of this, as shown in Figure 6-8, is to generate the horizontal series of circles over and over at different vertical positions, thus filling an entire rectangular area with rows and columns of circles.

The only other change we've had to make in the program is to the CIRCLE statement, where we've changed the fixed vertical coordinate from 10 to the counter variable *vert*.

When you run the program you'll see that it first draws the top row of circles, then the next row down, and so on down to the bottom row. What's happening is that the first time through the outer loop—when *vert*



```
FOR vert = 10 TO 240 STEP 20
   FOR horiz = 10 TO 170 STEP 20
      CIRCLE (horiz,vert), 10
   NEXT horiz
NEXT vert
```

FIGURE 6-8

Using nested loops to draw rows and columns of circles is 10—the inner loop is executed once. This draws the first row. The second time through the outer loop *vert* is 30 (10 plus a STEP of 20), so when the inner loop is executed again, the next row of circles is drawn with their centers at a vertical coordinate of 30. And so on, until the last line is drawn and the program ends.

This is an excellent program to step through to see how it works. Hold down the Command-T keys, or choose Trace On, to speed up the stepping process. The little box that outlines the statements will zip around the program listing, and the circles will be drawn one line at a time. You'll first see how nine circles are drawn by the inner loop, and then how the *NEXT vert* statement is executed, changing the vertical coordinate before the program goes back to draw another row.

It's important to notice that both the FOR and the NEXT of the inner loop must be inside the FOR and the NEXT of the outer loop. If you confuse this structure you'll be in big trouble: Your program won't work the way you want, and it will be difficult to figure out why. For instance, the following arrangement *won't* work:

```
FOR first = 1 TO 10
  FOR second = 1 TO 5
     :
     [other program lines]
     :
  NEXT first
NEXT second
```

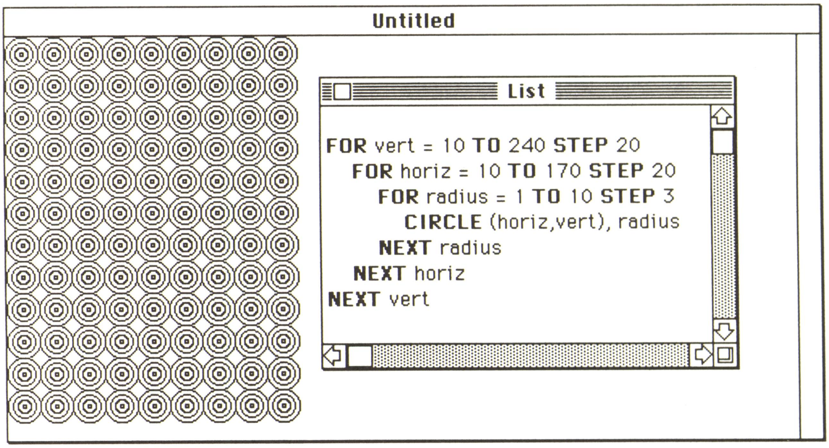
The FOR and NEXT that go with the counter variable *first* have a FOR *inside* their loop that goes with the counter variable *second*, but the NEXT that goes with *second* is *outside* the first loop. Disaster will result in the form of a FOR WITHOUT NEXT dialog box.

Notice that the indentation of the loops makes it easier to spot this sort of error. The FOR and NEXT on the same level of indentation should have the same counter variable. In the above program, they don't.

The third program, shown in Figure 6-9 on the following page, demonstrates that you can have more than one level of nested loops. In other words, you can nest a loop inside another loop (as we've done already) and then nest a third loop inside these two.

FIGURE 6-9

Columns and rows of concentric circles created with three nested loops



```
FOR vert = 10 TO 240 STEP 20
  FOR horiz = 10 TO 170 STEP 20
    FOR radius = 1 TO 10 STEP 3
      CIRCLE (horiz,vert), radius
    NEXT radius
  NEXT horiz
NEXT vert
```

This program draws the same rows and columns of circles as the last program did, but in addition draws concentric circles inside each of these circles. The two outer loops are the same as in the previous program. However, a new loop, with the counter variable *radius,* has been inserted inside the outer loops. This loop changes the radius so we can draw a number of circles at each circle location instead of only one as we did before. We've changed the CIRCLE statement to include the variable *radius* instead of using the fixed value 10 as in the last program.

Notice that there are now three levels of indentation, and that each FOR matches the NEXT directly below it.

An investment program

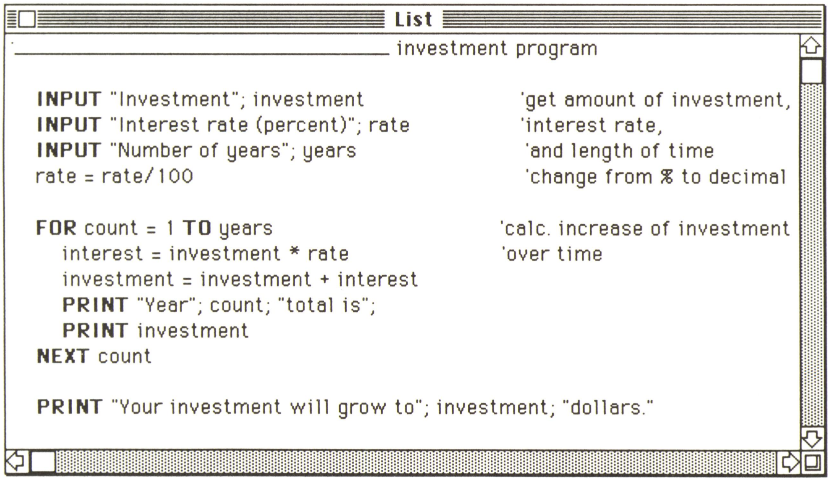
We are going to round out our discussion of FOR...NEXT loops a bit with a program that actually performs a useful function, and that uses a FOR...NEXT loop as an essential part of its structure.

Almost everyone has some money invested in a savings account, or in some other way that earns interest. The purpose of this program is to figure out how large an investment will have grown after a certain number of years, assuming that each year's interest is added to the original amount and that the total is then used to calculate the next year's interest.

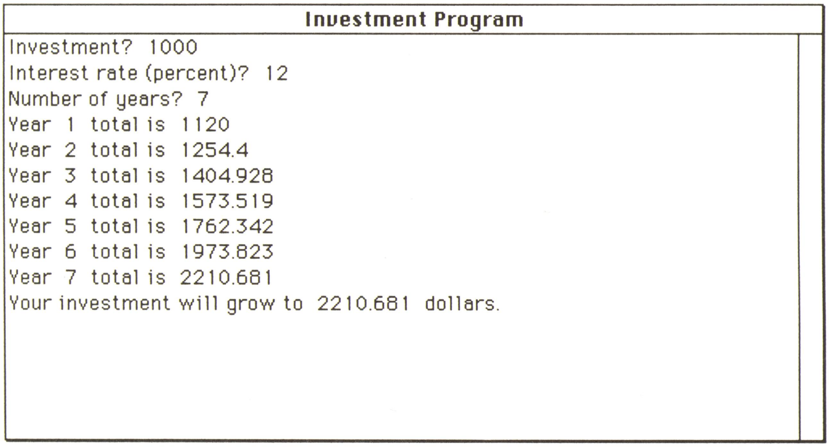
The user needs to let the program know how much money will be invested in the first place, and how much interest this money will be earning. We'll assume that the interest rate is expressed as a yearly percentage figure, and that the interest is compounded (added to the investment) only once a year. Finally, the user must tell the program how many years the money is going to be invested. The program and a sample of its output are shown in Figure 6-10.

FIGURE 6-10

A program to calculate an investment at the end of a specified time period



```
'________________________________ investment program

  INPUT "Investment"; investment                    'get amount of investment,
  INPUT "Interest rate (percent)"; rate             'interest rate,
  INPUT "Number of years"; years                     'and length of time
  rate = rate/100                                    'change from % to decimal

  FOR count = 1 TO years                           'calc. increase of investment
    interest = investment * rate                   'over time
    investment = investment + interest
    PRINT "Year"; count; "total is";
    PRINT investment
  NEXT count

  PRINT "Your investment will grow to"; investment; "dollars."
```



```
Investment? 1000
Interest rate (percent)? 12
Number of years? 7
Year  1  total is  1120
Year  2  total is  1254.4
Year  3  total is  1404.928
Year  4  total is  1573.519
Year  5  total is  1762.342
Year  6  total is  1973.823
Year  7  total is  2210.681
Your investment will grow to  2210.681  dollars.
```

Using INPUT statements, the program first finds out from the user the amount of the investment, the percentage rate, and the number of years the investment will continue. The rate is divided by 100 to change it from a percent to a decimal number. Then we enter a FOR...NEXT loop.

Using a variable limit in the FOR statement

There's something new in the way we use the FOR statement in this program. The upper limit is a variable, *years,* rather than a fixed number as in our previous examples. We can do this in a program as long as a value is assigned to the limit *outside* the loop, as it is here in the third INPUT statement, and not inside the loop, where BASIC would become confused. Since we want the calculation to be repeated once for each year the money is invested, and since we don't know until the user types it in how many years the money will stay invested, it's natural to use a variable in this situation.

Each year we'll take the original investment and multiply it by the interest rate to find out how much interest it has earned. Then we'll add this interest to the original investment. The next year, the investment will be larger; we'll multiply it by the same interest rate, obtaining a larger value of interest, which we'll again add to the investment. The program lines *interest = investment * rate* and *investment = investment + interest* perform these operations.

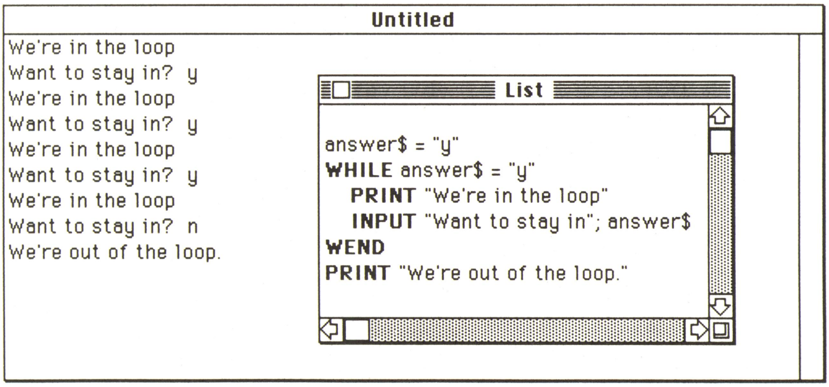
Each year we'll print out the number of the year (the counter variable *count*) and what our investment is that year. Finally, when we've done all the years and exit the loop, we'll print out the final amount, as shown in Figure 6-10. It's not absolutely necessary to print out the intermediate amounts each time through the loop, but it does make clear just how the program works.

You should now have an understanding of FOR...NEXT loops and how they operate. In the next section we'll talk about a loop structure that is similar to FOR...NEXT, but also has some important differences: the WHILE...WEND loop.

THE WHILE...WEND LOOP

The FOR...NEXT loop is great when we know, before we enter the loop, how many times we want BASIC to execute the statements inside it. If we want to do something ten times, we insert the line *FOR count = 1 TO 10* and that takes care of it. We can even set these limits within the program, as we saw in the investment program example above (provided we don't change them inside the loop itself).

But what happens if we don't know, at the time we start the loop, how many times we're going to execute it? Then the FOR...NEXT structure is not so easy to use. Instead, we'd use a somewhat different looping



```
We're in the loop
Want to stay in? y
We're in the loop
Want to stay in? y
We're in the loop
Want to stay in? y
We're in the loop
Want to stay in? n
We're out of the loop.
```

```
answer$ = "y"
WHILE answer$ = "y"
   PRINT "We're in the loop"
   INPUT "Want to stay in"; answer$
WEND
PRINT "We're out of the loop."
```

FIGURE 6-11

A simple WHILE...WEND loop program

structure called the WHILE...WEND loop. WHILE in this case means, "WHILE something is true, keep looping." WEND marks the end of the loop, and stands for "WHILE END."

Figure 6-11 shows an example of a simple program that uses a WHILE...WEND loop. Once into the loop, this program asks whether the user wants to stay in the loop. If the answer is a lowercase *y*, the program executes the loop again, printing *We're in the loop.* If the user types anything other than *y*, the program jumps out of the loop, prints *We're out of the loop,* and ends.

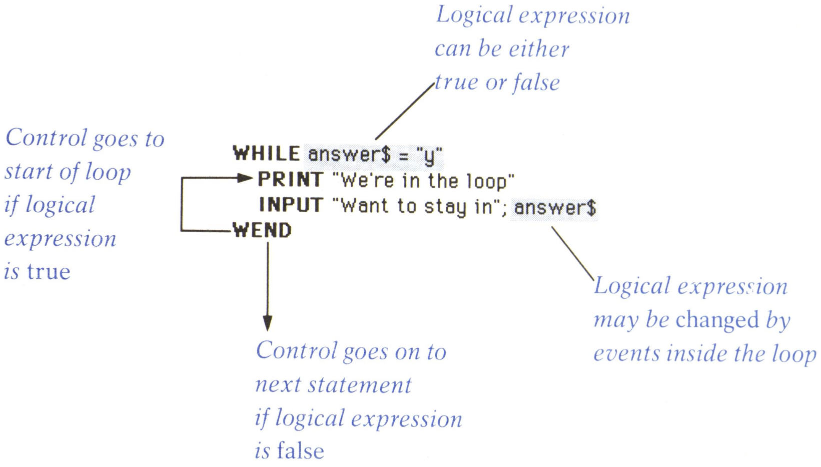
Notice that this sort of program could not have been written using a FOR...NEXT loop. It's not the completion of a certain *number* of loops that causes us to exit from the loop, it's the fact that a certain *condition* becomes true: in this case the condition that the variable *answer$* is no longer equal to *y.*

Structure of the WHILE...WEND loop

The WHILE statement (like the FOR in a FOR...NEXT loop) marks the beginning of the loop, while WEND (like the NEXT in a FOR...NEXT loop) marks the end. The major difference between WHILE...WEND and FOR...NEXT lies in what follows the WHILE. Instead of a counter variable and a range of values, we have a single *logical expression.* While this expression is true, the WEND statement causes the program to jump to the

FIGURE 6-12

The structure of the WHILE... WEND loop



```
WHILE answer$ = "y"
  PRINT "We're in the loop"
  INPUT "Want to stay in"; answer$
WEND
```

Logical expression can be either true or false
Control goes to start of loop if logical expression is true
Logical expression may be changed by events inside the loop
Control goes on to next statement if logical expression is false

start of the loop, and when the expression becomes false WEND exits the loop and then goes on with the rest of the program. This is shown above in Figure 6-12.

Logical expressions

What's a *logical expression?* It's simply a statement that can be either true or false. Thus in the present example, the expression *answer$ = "y"*, which follows the WHILE in the statement *WHILE answer$ = "y"* may be true, or it may not. If *answer$* is *n*, or any other letter, or no letter at all, then the statement is false. If *answer$* is in fact equal to *y*, then the statement is true. A logical expression can change from true to false, or from false to true, during the course of the program. In this case, we make sure that it's true at the beginning of the loop by executing the statement *answers$ = "y"*.

In the body of the loop the truth of the expression may change or not, depending on the user's answer to the question, *Want to stay in?* If the user types *y*, the logical expression continues to be true. If the user types anything else, the expression becomes false, causing the WEND statement to stop looping.

Expressions versus assignment statements

You have probably noticed that the statement *answer$* = "*y*" occurs twice in the program, and that it's *used in entirely different ways* in these two cases. In the first line of the program it's used as an assignment statement. We're already familiar with assignment statements, from such earlier examples as *radius* = *50*, which takes the variable *radius* and assigns it a value of 50. Similarly, the first line of our present example, *answer$* = "*y*", takes the string value *y* and assigns it to the variable *answer$*.

This is very different from the logical expression *answer$* = "*y*" that follows WHILE. This logical expression doesn't change anything, it just sits there waiting to be evaluated as either true or false. The WEND statement, when it is executed, looks at the logical expression and decides whether to execute the loop again, depending on whether the expression is true or false.

The difference between an assignment statement and a logical expression is that the equal sign is really used in two different ways. In an *assignment statement* the equal sign means, "Give the variable on the left of the equal sign the value of the expression or constant on the right." In a *logical expression* the equal sign is used in the way we're familiar with from algebra: to indicate that two things are equal.

The concept of logical expressions may become clearer when we explore *relational operators,* since the equal sign in a logical expression is an example of a relational operator.

Relational operators

Relational operator is a high-falutin' term for what is really a fairly simple concept. Relational operators are simply ways of comparing two things, in the same way we compare the cost of apples or the weight of linebackers. For instance, we might say "The cost of a Cadillac is greater than the cost of a Toyota." The phrase *greater than* acts as a relational operator. If we said, "The weight of this Toyota is equal to the weight of the Pittsburgh Steelers' offensive line," the phrase *is equal to* acts as a relational operator. Relational operators express statements that are either true or false. It is either true or false that a Cadillac costs more than a Toyota, and it is either true or false that the weight of this Toyota is equal to the weight of the Pittsburgh Steelers' offensive line.

In BASIC we express these relational operators in a sort of shorthand, so they won't take up so much room:

= means *is equal to*
<> means *is not equal to*
< means *is less than*
> means *is greater than*
<= means *is less than or equal to*
>= means *is greater than or equal to*

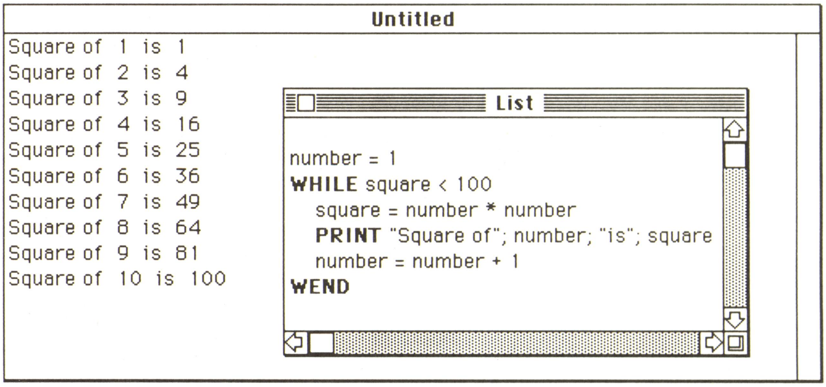
Using a relational operator in a program

Let's look at an example of a program that uses one of these relational operators in a WHILE...WEND loop, as shown in Figure 6-13.

This program calculates the squares of numbers starting with 1, and stops when the square of the number (not the number itself) is 100. As you probably know, the square of a number is simply the number multiplied by itself, and the number whose square is 100 is 10. So we could have written this program with the FOR...NEXT loop *FOR number = 1 TO 10* and so forth. But suppose we wanted to calculate all the squares until the square was 65,536 or some other large number? The structure of this WHILE...WEND loop figures out when to stop looping for us.

FIGURE 6-13

Using a relational operator in a WHILE...WEND loop



```
Square of  1  is  1
Square of  2  is  4
Square of  3  is  9
Square of  4  is  16
Square of  5  is  25
Square of  6  is  36
Square of  7  is  49
Square of  8  is  64
Square of  9  is  81
Square of  10  is  100
```

```
number = 1
WHILE square < 100
   square = number * number
   PRINT "Square of"; number; "is"; square
   number = number + 1
WEND
```

Our program starts with *number* equal to 1; then, inside the loop, it takes *number* and multiplies it by itself to find the square, and prints out the result. Finally, it adds 1 to *number,* so that the next time through it will be finding the square of the next higher number.

How does WEND know when to stop looping? It looks up at the logical expression *square* $<$ *100* in the WHILE statement to see if it is still true. And what does this logical expression mean? Translated into English, it says *square is less than 100.* Is it true or false? That depends on the value of *square,* of course. The first time through the loop *square* is 1 times 1, or 1. That's less than 100, so the expression is true, and the loop is executed again. This time, *number* is 2, so *square* is 2 times 2, or 4. That's still less than 100. It's not until *number* is 10 that *square* is no longer less than 100. (In fact, it's exactly 100.) At that point the WEND statement sees that the logical expression *square* $<$ *100* is no longer true, and it ends the loop, thus terminating the program.

We'll see other uses for relational operators in later chapters, notably in the section on IF...THEN statements in Chapter 7.

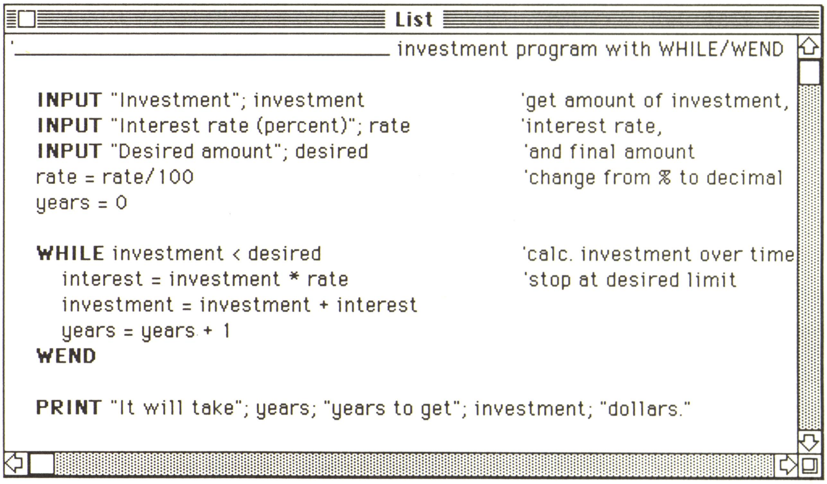
Another investment program

Let's round off our discussion of WHILE...WEND loops with another financial program. This one tells us how long it will take for an investment to grow to a certain amount, given an initial amount and the interest rate. As in the investment program we created in Figure 6-10, we assume that the interest is compounded once a year.

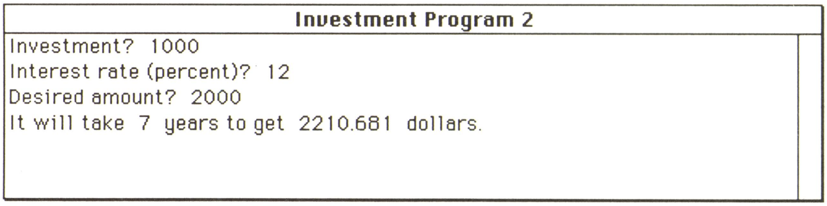
Notice how this program differs from the one in Figure 6-10, which used a FOR...NEXT loop. In the present example, we don't know in advance how many times the program will go through the loop, since we don't know how long it will take an investment to grow to the amount specified. Thus the WHILE...WEND loop is a more appropriate structure. The program with a sample output is shown on the next page, in Figure 6-14.

FIGURE 6-14

A program to calculate the time needed to reach an investment goal



```
'__________________________________ investment program with WHILE/WEND

  INPUT "Investment"; investment                  'get amount of investment,
  INPUT "Interest rate (percent)"; rate           'interest rate,
  INPUT "Desired amount"; desired                 'and final amount
  rate = rate/100                                 'change from % to decimal
  years = 0

  WHILE investment < desired                      'calc. investment over time
     interest = investment * rate                 'stop at desired limit
     investment = investment + interest
     years = years + 1
  WEND

  PRINT "It will take"; years; "years to get"; investment; "dollars."
```



```
Investment?  1000
Interest rate (percent)?  12
Desired amount?  2000
It will take  7  years to get  2210.681  dollars.
```

As you can see, at 12 percent interest it will take 7 years to double the investment. The program is similar to the earlier one, except that the WHILE...WEND loop continues to loop until the variable *investment* exceeds the amount of the variable *desired.* The variable *years* has been counting the number of times through the loop, and at this point the program exits from the loop and prints out the number of years.

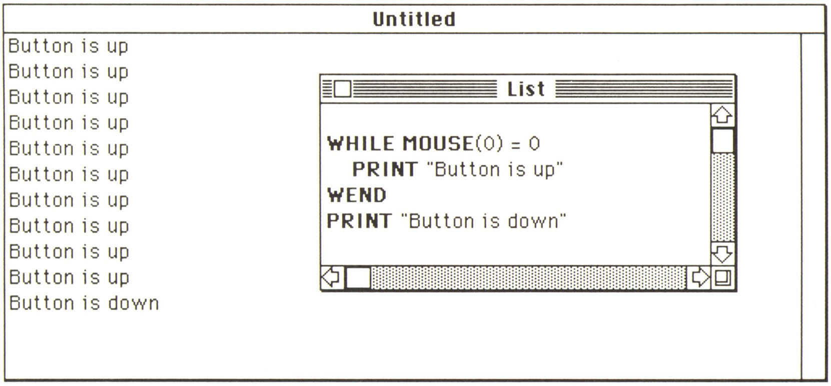
PROGRAMMING WITH THE MOUSE

Now that you have been introduced to WHILE...WEND loops, we can explore one of the Macintosh's most exciting features, the mouse. Knowing how to interact with the mouse can add an exciting new dimension to your programming. You'll be able to create BASIC games and other programs that use the mouse for input, as do MacWrite, MacPaint, and almost all of the other Macintosh programs. Besides learning about the mouse in this section, you'll also be learning more about WHILE...WEND loops and how they work.

In the following examples we'll learn two things about using the mouse: how to tell when the mouse button is pushed down, and how to tell where the mouse pointer is in the Output window.

FIGURE 6-15

The mouse button program



```
Button is up
Button is up
Button is up
Button is up
Button is up
Button is up
Button is up
Button is up
Button is up
Button is up
Button is down
```

```
WHILE MOUSE(0) = 0
  PRINT "Button is up"
WEND
PRINT "Button is down"
```

Is the mouse button down?

Figure 6-15 displays a program that figures out if the mouse button is down. It does so by using a WHILE...WEND loop, and a curious-looking expression, MOUSE(0). MOUSE(0) is an example of a *function*. We'll be encountering a lot of functions from now on, so this is a good time say something about what they are.

BASIC functions

Functions can be thought of as the private detectives of programming. We use them to find out something for us, in the same way we might hire a detective to obtain a certain piece of information. We need to select the right function for the job, just as we would need to hire the right detective: We wouldn't ask a specialist in marital problems to handle an industrial espionage case, for instance. We need to tell the function what we need to find out, as we would tell the detective that we need him to find a missing person. Then the function goes off, finds the information, and returns it to the program.

Of course, the kind of information we ask functions to return is not usually as dramatic as that often obtained by detectives in books and on TV. However, functions can generate a certain amount of low-key excitement, and using them is essential to obtaining certain kinds of information for our program.

Functions are one of the two classes of BASIC keywords: Statements are the other. So far the BASIC keywords we've discussed, like PRINT, CIRCLE, WHILE...WEND, and so forth have all been *statements*. A statement actually *does* something. It carries out an action, like printing something, drawing a circle, or causing the program to loop. It essentially tells BASIC, "Do this."

A function plays a less active role. It is an information gatherer. A function must always be used in an expression that can use the information, like an assignment or PRINT statement. A function is treated like a *value.* When our program executes a function, the value that the function finds is said to be *returned* to the program. Executing a function is often referred to as *calling* a function. Our program calls a function, which returns a value to the program. Let's look at our example program in Figure 6-15 and see how the function MOUSE(0) is used in it.

The first thing you'll notice about MOUSE(0) is the 0 inside the parentheses. What does this mean? Putting something in parentheses at the end of a function is the standard way of telling the function what it's supposed to use for input. There are, for example, a number of different things we can do with the MOUSE function. The 0 in parentheses tells MOUSE, "Find out whether the button is up or down."

You will notice that MOUSE(0) is used in a logical expression. MOUSE(0) can return a number of different values, depending on what is happening to the mouse button. If it returns a value of 0, the mouse button is not down and has not been pressed since the last time the program called a MOUSE(0) function. If the MOUSE(0) returns a 1, the mouse button was clicked once since the last MOUSE(0) call. We will learn much more about MOUSE(0) in the next chapter, but if you're curious, a complete table of the values it can return, and of the other mouse functions, can be found under "Mouse" in the Microsoft BASIC manual.

In our program, if MOUSE(0) returns a 0, we know the mouse button has not been pressed. The logical expression following WHILE is therefore true, and the loop keeps looping, printing out *Button is up* on the screen. As soon as the button is pressed, the logical expression becomes false, WEND terminates the loop, the phrase *Button is down* is printed, and the program terminates.

Where is the mouse?

The function MOUSE(1) returns the x coordinate, or horizontal position, of the mouse pointer and MOUSE(2) returns the y coordinate, or vertical position. Knowing where the mouse pointer is can be very useful information for our program, as we'll see in future chapters.

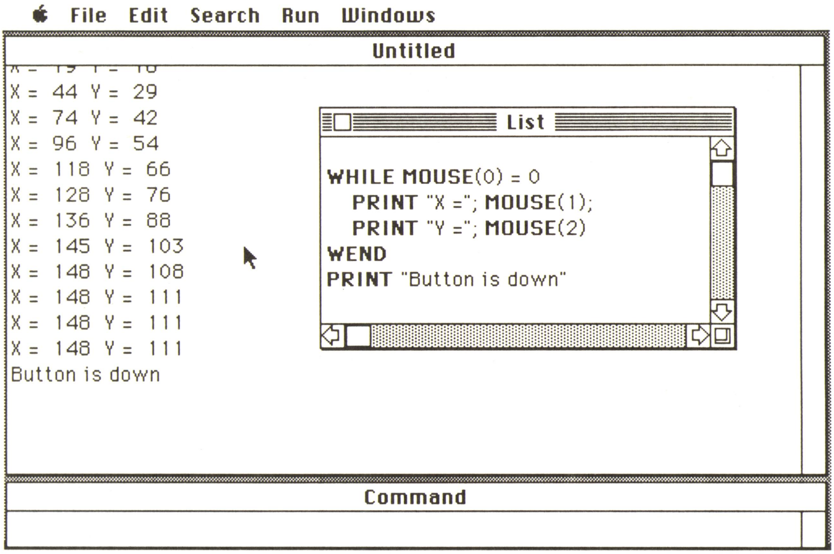
There is an oddity about exactly which mouse-pointer position MOUSE(1) and MOUSE(2) return. They don't, as you might expect, tell you where the pointer *is* at the exact time they themselves are executed.

Instead, they let you know where the mouse pointer *was* when the last MOUSE(0) function was executed. So if our program wants to find out where the mouse pointer is at a precise moment, it must first execute a MOUSE(0), even though it may not care whether the mouse button is up or down, and then immediately call MOUSE(1) and MOUSE(2) to find the pointer's location.

Figure 6-16 shows a program that continuously finds out where the mouse pointer is, and prints out this information. Since the program must call MOUSE(0) each time the loop is executed to see if it should exit from the loop, the mouse-pointer position returned by MOUSE(1) and MOUSE(2) is kept up to date. When we click the mouse button, the logical expression in the WHILE statement is no longer true, the loop is terminated, and the program prints out the *Button is down* phrase and halts.

This is a very interesting program to type in and run, since it lets you explore BASIC's graphics-coordinate system. Move the pointer anywhere on the screen, and see if the coordinates that you find there are what you expected.

This brief introduction to the mouse has only touched the surface of what it can do for our BASIC programs on the Macintosh. We'll learn many more of its capabilities in future chapters.



```
X = 44  Y = 29
X = 74  Y = 42
X = 96  Y = 54
X = 118  Y = 66
X = 128  Y = 76
X = 136  Y = 88
X = 145  Y = 103
X = 148  Y = 108
X = 148  Y = 111
X = 148  Y = 111
X = 148  Y = 111
Button is down
```

```
WHILE MOUSE(0) = 0
  PRINT "X ="; MOUSE(1);
  PRINT "Y ="; MOUSE(2)
WEND
PRINT "Button is down"
```

FIGURE 6-16

A mouse-pointer location program

THE GOTO STATEMENT AND LABELS

We have deliberately avoided discussing the GOTO statement until this point in the book. GOTO is a somewhat controversial statement. It can be very useful in certain situations, but it can also get us into a lot of trouble.

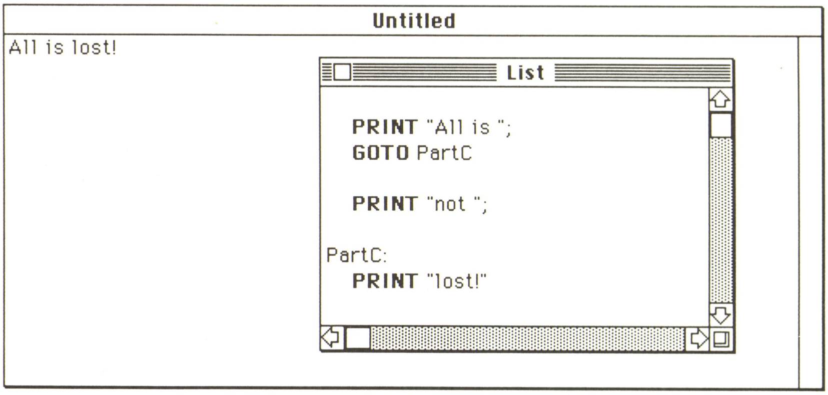
The problem with GOTO is that it's almost too easy to use. It's easy to understand, and many programmers, lulled into a false sense of security by GOTO's simplicity, use it all over their programs whether it's appropriate or not. The structure of such a program soon resembles the structure of a bowl of spaghetti, with the result that the programmer loses track of what's going on and must abandon the program in despair.

For this reason, GOTO should be used sparingly, if at all. We are introducing it here because it can be useful in some situations, and because it serves as a natural introduction to the topic of *labels.*

What does GOTO do? You're already familiar with loops, in which the flow of a program jumps back from a NEXT statement or a WEND statement to the beginning of a loop. GOTO performs a similar sort of jump, except that it can cause the program to jump *anywhere* in the program that we specify. How do we tell GOTO where to jump? By using a *label* at a certain point in the program, and then using the label in a GOTO statement.

Demonstrating GOTO and labels

Figure 6-17 is a very short program that demonstrates how GOTO and labels work together. For clarity, this program is divided into three parts by blank lines, and all the lines except the label *PartC* have been indented. You can think of the three parts as Part A, Part B, and Part C.



```
All is lost!
```

```
   PRINT "All is ";
   GOTO PartC

   PRINT "not ";

PartC:
   PRINT "lost!"
```

FIGURE 6-17

A GOTO demo program

In Part A the program prints *All is,* and then it encounters the *GOTO PartC* statement. This statement does just what it describes: It directs the program to go directly to the place in the program labeled *PartC*. Looking at the listing, we see that this is the next-to-last line in the program. So the program goes to the *PartC* label, executes the next statement, which is *PRINT "lost!"*, and then ends. Note that Part B, which is the statement *PRINT "not "*, is never executed. The GOTO caused the program to *skip over* this statement.

BASIC labels

A label is a name we give to a place in our program. It is usually written on a line by itself, as shown in the previous example, but it can also precede a statement, as in

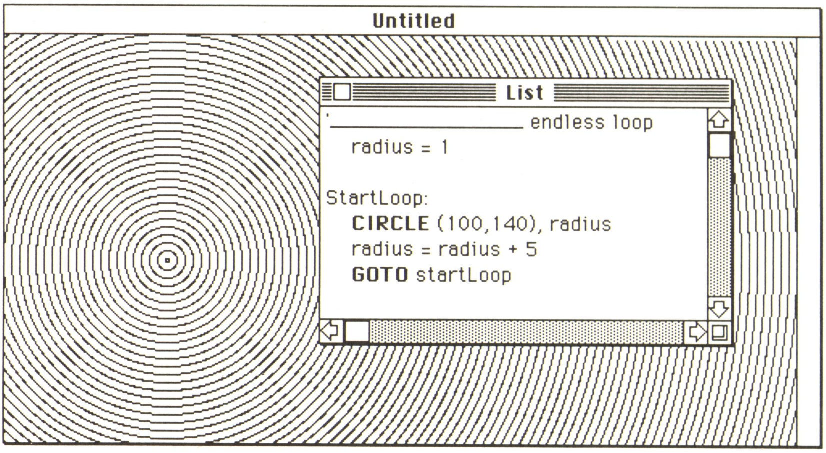
```
PrintPlace2: PRINT "Here we are at PrintPlace2"
```

The label must always be followed by a colon. But when the label is referred to in a GOTO or other statement, the colon is *not* used. The rules for making up label names are the same as those for variable names. It must be 40 or fewer characters long; it must start with a letter; and it can contain letters, numbers, and periods. There can't be any spaces in the label name, and there can't be a space between the name and the colon. We can use upper- or lowercase letters as we wish, but BASIC can't tell the difference—label, Label, and LABEL are all the same as far as it's concerned.

Line numbers

We don't recommend using line numbers, but you may come across them in pre-Microsoft BASIC programs, so we'll mention them here.

Line numbers are similar to labels, except that they are written without a colon at the end. They can range from 0 to 65,529, and like labels they're used to mark a place in the program so we can jump to them from other parts of the program. Because line numbers are much less informative about what a particular part of the program does than well-chosen labels, there's really not much need to use them.



```
'_______________ endless loop
  radius = 1

StartLoop:
  CIRCLE (100,140), radius
  radius = radius + 5
  GOTO startLoop
```

FIGURE 6-18

A GOTO loop

Looping with GOTO

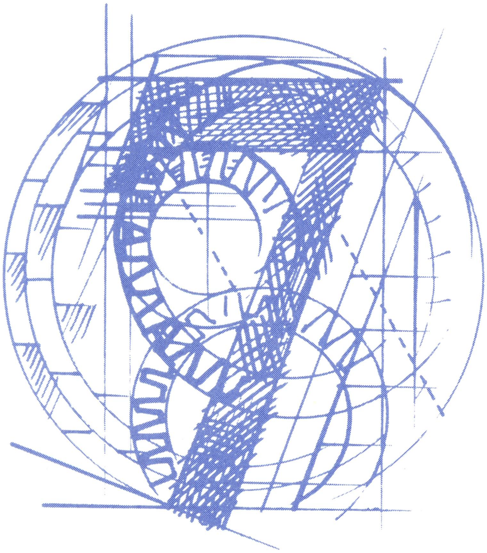
We don't always have to jump forward in our program with GOTO; we can also jump backwards. This can create a loop, as shown in Figure 6-18.

This program has the label *StartLoop* as its second line, and the statement *GOTO StartLoop* as the last line. These two lines form the start and end of a loop. The program first draws a circle with a radius of 1. Then it increments the radius by 5, and jumps back to *StartLoop.* It now draws another circle, larger than the first, increments the radius, jumps back to *StartLoop,* and so forth, over and over.

We have created a program that will never end; it will simply go on drawing larger and larger circles, as displayed in the Output window in Figure 6-18. To stop this program we will need to choose Stop from the Run menu. There is a way to exit from a GOTO loop such as this from within the program, but it involves the IF...THEN statement, which is the subject of the next chapter.

REFRESH-YOURSELF WINDOW

In this chapter we've learned about three different kinds of loops: FOR...NEXT, WHILE...WEND, and GOTO. We've also been introduced to relational operators like "less than" and "greater than," we've discovered something about how to use the mouse in our programs, and we've learned about labels. The next chapter will build on this knowledge and introduce the idea of decision-making.



Decisions

This chapter focuses on yet another important BASIC concept: decision making. *To extend the analogy we have been using in the last few chapters, if variables are the "soul" of BASIC, and loops are its "muscle," then we might say that decisions are its "intelligence." Loops let you do the same thing over and over again, while decisions give your program the power to react* intelligently to changing circumstances. Taken together, variables, loops, and decisions give BASIC programs great power and flexibility.

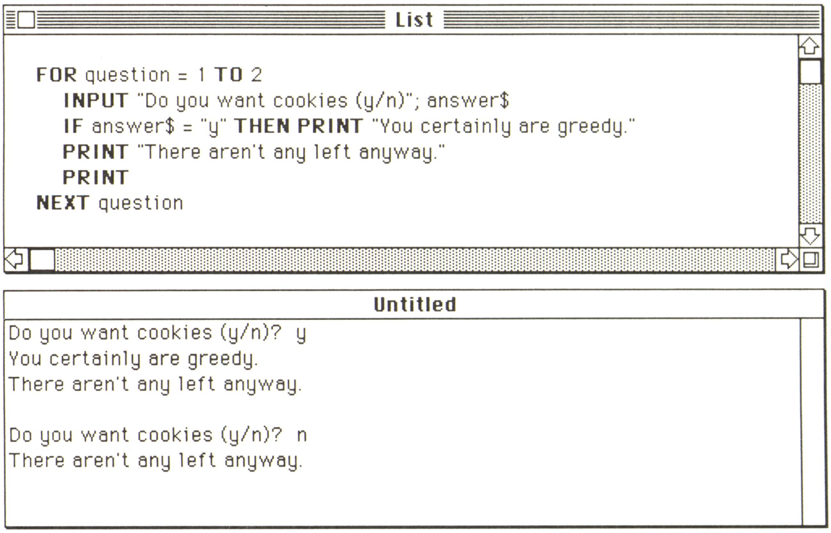
The heart of decision making in BASIC is the IF...THEN statement. We'll explore this statement, and its more sophisticated cousin, the IF...THEN...ELSE statement. We will also introduce the idea of *logical operators,* which increase the power of IF...THEN statements.

The mouse provides an interesting, interactive way to demonstrate the programming ideas involved in decision making. Also, the mouse is central to programming the Macintosh. For these reasons, many examples in this chapter make use of the mouse and graphics, and we'll touch on some of the more advanced ways we can use the mouse.

THE IF...THEN STATEMENT

The IF...THEN statement describes itself very well. *If* something is true, *then* the statement causes something to happen. Figure 7-1 shows a simple program that uses an IF...THEN statement.

This program, when run, first asks us, *Do you want cookies (y/n)?* Our answer, which should be either *y* or *n,* is assigned to the string variable *answer$*. Next, the IF of the IF...THEN statement looks at the logical expression *answer$ = "y"*. This expression can either be true, or it can be false. If it's true—that is, if we typed *y*—then the statement following THEN is executed, which in this case means that the phrase, *You certainly are greedy,* is printed. If the logical expression is false—that is, if we typed



```
FOR question = 1 TO 2
   INPUT "Do you want cookies (y/n)"; answer$
   IF answer$ = "y" THEN PRINT "You certainly are greedy."
   PRINT "There aren't any left anyway."
   PRINT
NEXT question
```

```
Do you want cookies (y/n)?  y
You certainly are greedy.
There aren't any left anyway.

Do you want cookies (y/n)?  n
There aren't any left anyway.
```

FIGURE 7-1

A program using the IF...THEN statement anything other than *y*—then the statement following THEN is *not* executed. (In either case, the last line of the FOR...NEXT loop, which prints *There aren't any left anyway,* is always executed.) The Output window in Figure 7-1 shows how both possible answers to the INPUT prompt change the behavior of the program.

The IF...THEN statement has four parts:

IF *logical expression* **THEN** *do statement*

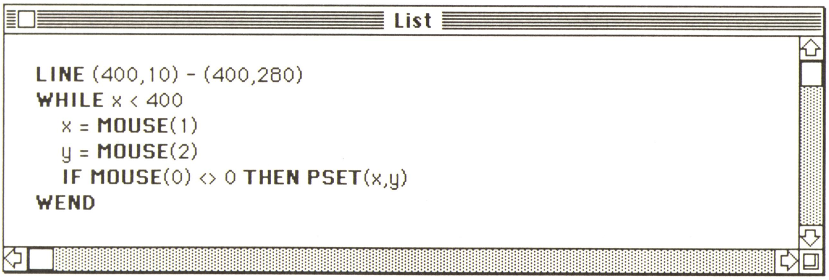
The *logical expression* can use any of the relational operators described in the last chapter. Examples might be *a* < *b, savings* = *investment,* and so on. The logical expression can also make use of *logical operators,* which we'll explore later on in this chapter. The *do statement* can be any valid BASIC statement. Examples would be *PSET(x,y),* and *GOTO ElseWhere.* (Note, however, that the key word GOTO is optional in IF...THEN statements: If BASIC encounters a label immediately following THEN, it assumes we want to GOTO the label.)

That's really all there is to IF...THEN. If the logical expression following the IF is true, the statement following the THEN is executed; otherwise it isn't. This structure may appear simple, but it is a powerful and essential part of almost any useful BASIC program.

IF...THEN decisions with the mouse

As another example of using an IF...THEN statement, let's write a program that lets us draw pictures in the Output window using the mouse. The program and sample doodle are shown in Figure 7-2.

The *WHILE x < 400* loop lets us draw only on the left side of the line drawn by the LINE statement. This gives us a convenient way to exit the program: As soon as we push the pointer over to the right side of the line, the logical expression in the WHILE statement is no longer true, the loop ends, and the program is over. As long as the pointer stays on the left side of the line, the program continues to operate.



```
LINE (400,10) - (400,280)
WHILE x < 400
   x = MOUSE(1)
   y = MOUSE(2)
   IF MOUSE(0) <> 0 THEN PSET(x,y)
WEND
```

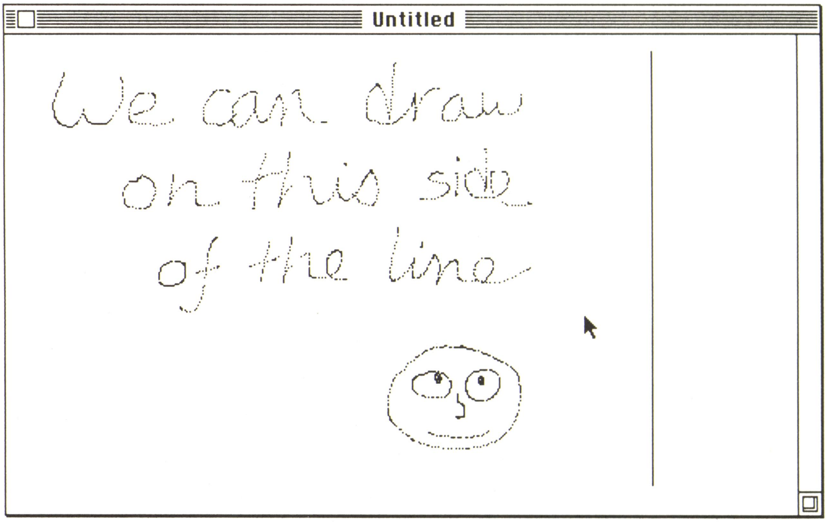
FIGURE 7-2

A mouse-drawing program

(continued)

FIGURE 7-2

(continued)



Notice the logical expression MOUSE(0) <> 0 in the IF...THEN statement, and recall from our discussion of relational operators in the last chapter that <> means "not equal to." This expression is true only when the mouse button is down, since when it is up MOUSE(0) is equal to 0.

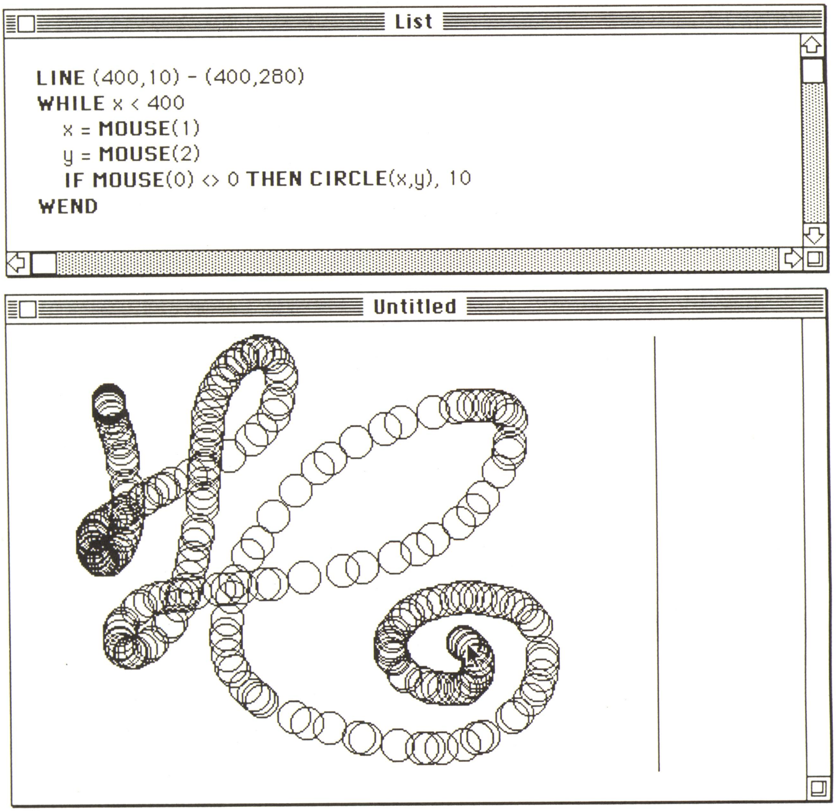
Inside the WHILE...WEND loop the program constantly reads the x and y coordinates of the mouse pointer's location with MOUSE(1) and MOUSE(2). *If* the mouse button is down—that is, if the logical expression MOUSE(0) <> 0 is true—*then* a pixel is drawn on the screen at the location of the pointer by the statement PSET(x,y). But if the mouse button is *not* down, the logical expression is therefore false, the statement following THEN is *not* executed, and no pixel is drawn.

Notice that the first time the loop is executed, we ask for an x and y position before we get to the statement containing MOUSE(0). In the last chapter, we said that MOUSE(1) and MOUSE(2) need to be preceded by MOUSE(0). Since the WHILE loop repeats continuously, the current position of the mouse pointer is set by MOUSE(0) *just before* the program returns to the beginning of the loop.

You'll notice that we can change the density of the line—that is, how closely spaced the dots are—by moving the mouse at different speeds. The slower we move it, the denser the line is. We can use this fact to achieve different effects in our drawings.

FIGURE 7-3

A circle-drawing program

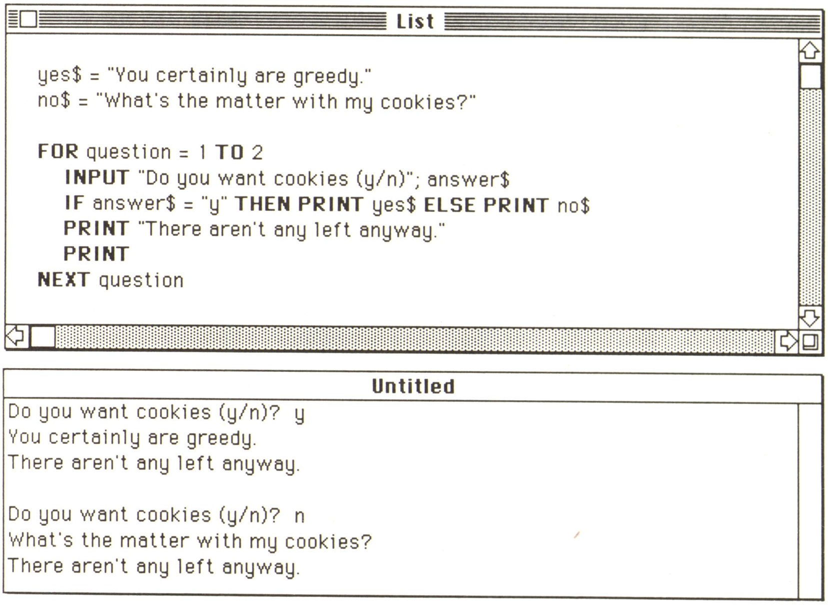


```
LINE (400,10) - (400,280)
WHILE x < 400
   x = MOUSE(1)
   y = MOUSE(2)
   IF MOUSE(0) <> 0 THEN CIRCLE(x,y), 10
WEND
```

But we're not limited to drawing only single-pixel lines with the mouse and IF...THEN: We can also draw with *shapes* such as rectangles and circles. Figure 7-3 shows a short program that draws with circles. This program uses the same technique, including the IF...THEN statement, as the preceding one.

THE IF... THEN...ELSE STATEMENT

In an IF...THEN statement, the action after the THEN is done only if the logical expression is true; if it isn't true, the action is not done. But sometimes it's convenient to have the program do one thing if the expression is true, and *something else* if it isn't. The IF...THEN... ELSE statement was created to deal with this situation. Figure 7-4 on the next page shows our "cookie" program with an IF...THEN...ELSE statement substituted for the original IF... THEN statement.



```
yes$ = "You certainly are greedy."
no$ = "What's the matter with my cookies?"

FOR question = 1 TO 2
   INPUT "Do you want cookies (y/n)"; answer$
   IF answer$ = "y" THEN PRINT yes$ ELSE PRINT no$
   PRINT "There aren't any left anyway."
   PRINT
NEXT question
```

```
Do you want cookies (y/n)?  y
You certainly are greedy.
There aren't any left anyway.

Do you want cookies (y/n)?  n
What's the matter with my cookies?
There aren't any left anyway.
```

FIGURE 7-4

A program using the IF... THEN... ELSE statement

This program behaves the same as the earlier one, except when we type *n* in answer to the cookie question. In this case it prints out *What's the matter with my cookies?* Thus it has a response to *both* possible answers, instead of only one.

Besides adding the ELSE to the program, we've made another change so as to avoid having an excessively long program line in the IF... THEN... ELSE statement. We've assigned the phrases *You certainly are greedy* and *What's the matter with my cookies?* to the variables *yes$* and *no$,* respectively. This doesn't change the program's operation; it merely makes this line a more manageable length:

```
IF answer$ = "y" THEN PRINT yes$ ELSE PRINT no$
```

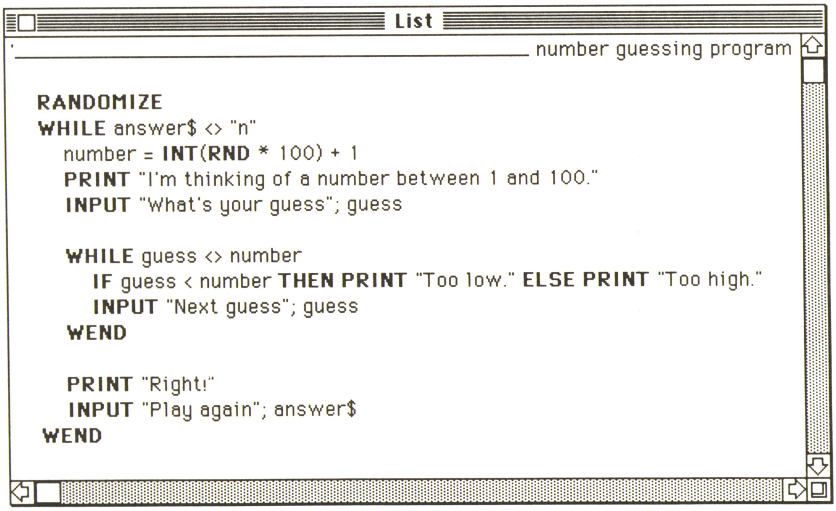
As in the earlier cookie program, the logical expression following IF is *answer$ = "y"*. However, there are now two possible statements that can be executed depending on the value of this expression. If it's true, then the phrase in the string variable *yes$* is printed. But if the expression is false, the phrase in the variable *no$* is printed. The general structure of all IF... THEN... ELSE statements is:

> **IF** *logical expression* **THEN** *do first statement* **ELSE** *do second statement*

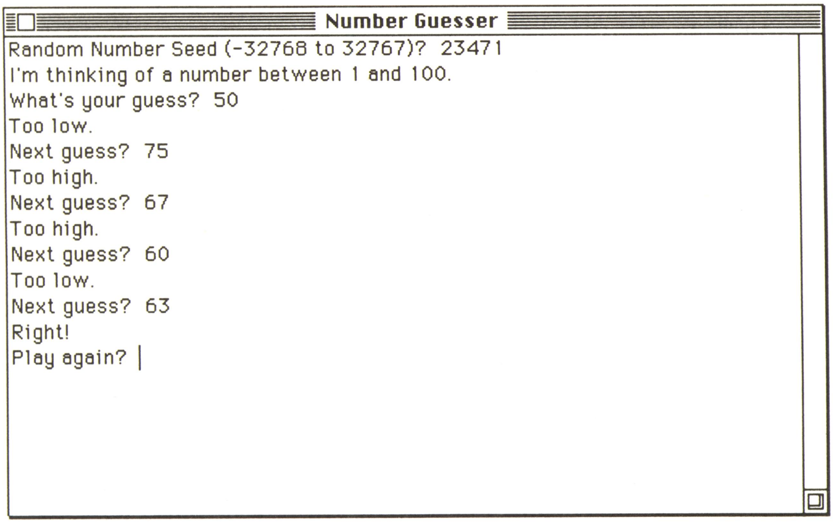
RANDOM NUMBERS: RND AND RANDOMIZE

Let's program a game that puts the power of the IF...THEN...ELSE statement to work. This program thinks of a number, and then lets the user guess what the number is. It tells you if the number is too high or too low, and of course it tells you if you get it right. The program listing and a sample run are shown in Figure 7-5.

Besides offering another chance to practice using IF...THEN...ELSE, this program introduces the topic of *random numbers.* What are random numbers? As the name implies, they are numbers whose value you don't know in advance. Their purpose is to introduce an element of *chance* into what would otherwise be a totally predictable machine. They are



```
RANDOMIZE
WHILE answer$ <> "n"
  number = INT(RND * 100) + 1
  PRINT "I'm thinking of a number between 1 and 100."
  INPUT "What's your guess"; guess

  WHILE guess <> number
    IF guess < number THEN PRINT "Too low." ELSE PRINT "Too high."
    INPUT "Next guess"; guess
  WEND

  PRINT "Right!"
  INPUT "Play again"; answer$
WEND
```



```
Random Number Seed (-32768 to 32767)?  23471
I'm thinking of a number between 1 and 100.
What's your guess?  50
Too low.
Next guess?  75
Too high.
Next guess?  67
Too high.
Next guess?  60
Too low.
Next guess?  63
Right!
Play again? |
```

FIGURE 7-5

The listing and a sample run of the number-guessing program useful in a variety of game programs such as this, but they also have serious uses in providing test input for programs and for modeling various statistical processes.

For the time being, we will just assume that the program line *number= INT(RND * 100) +1* generates a random number between 1 and 100. We'll discuss later how it does it.

Once it has generated a random number, the program asks the user to guess the number. Assuming that the guess is wrong, the program enters the inner WHILE...WEND loop, and uses the IF...THEN...ELSE statement to tell the user whether the guess is less than the number, or greater. Then the user is asked for another guess. The program cycles around the inner WHILE...WEND loop until the guess is the same as the number; then it exits from the loop, prints *Right!* and asks if the user wants to play again. The answer to this question determines whether the outer WHILE...WEND loop is executed again.

Let's turn now to the question of how the program generates random numbers.

The RND function

Random numbers are generated in BASIC by the RND function. This function returns a random value between 0 and 1. Actually the upper limit isn't exactly 1, it's .9999999999999999. This is a number 16 digits long! A typical number returned by RND might be 0.8762348875347342, for example. However, we don't really want all these digits. In our program, what we want is a simple whole number (technically called an *integer*) between 1 and 100. We must therefore go to a little trouble to convert the number returned by RND to the form we want.

We first multiply the number by 100, so it will be in the range 0 to 99.99999999999999. Then we need to get rid of all those objectionable numbers to the right of the decimal point. We do that with another function called *INT.*

The INT function

INT stands for INTeger. The purpose of this function is to get rid of all the numbers to the right of the decimal point; that is, to turn a number into an integer. If we executed the program line *PRINT INT(12.75),* for example, the result would be 12, which is the integer part of 12.75.

Notice that the number we want to apply the function *to* goes in parentheses following the INT. The resulting numerical value is returned by the function itself, as if it were a variable with that value.

The expression *RND * 100* in our program will be returning a number like 57.3692735183762. We want to turn this into 57, so we apply the INT function. INT(57.3692735183762) is 57, so the expression INT(RND * 100) gives us just what we want: a random integer between 0 and 99. Well, almost what we want. We would really like the numbers to range from 1 to 100, since that's a traditional range for games of this sort. So we add 1 to the result of the INT function, which will allow the statement to produce 100 occasionally. And that's how we arrive at the actual line shown in the program: *number = INT(RND * 100) + 1.*

There's only one remaining problem with RND that needs to be dealt with: Unfortunately, RND always generates *the same series* of random numbers. Without RANDOMIZE, every time we start BASIC and run this program, the first random number generated by the program is 13, the next one is 66, and so on. This isn't appropriate for the present example, since it's no fun to play the game if we already know all the answers. How can we get RND to generate *different* random numbers each time we start the program?

The RANDOMIZE statement

The answer to this problem is the RANDOMIZE statement. When executed, this statement, as you can see in the sample run of the number-guessing program, asks us to type in a number between −32768 and 32767. BASIC then uses the number we type in as a "seed," or starting point, to generate a series of random numbers. Of course, if we type in the same number each time, the same sequence of random numbers will be generated, so we need to type in a different number each time we start the program. The resulting series of numbers will have no obvious relationship to the number we typed in.

Notice that we can play the game over and over without typing in a new seed; it's only when we *start* the program that we need to use RANDOMIZE.

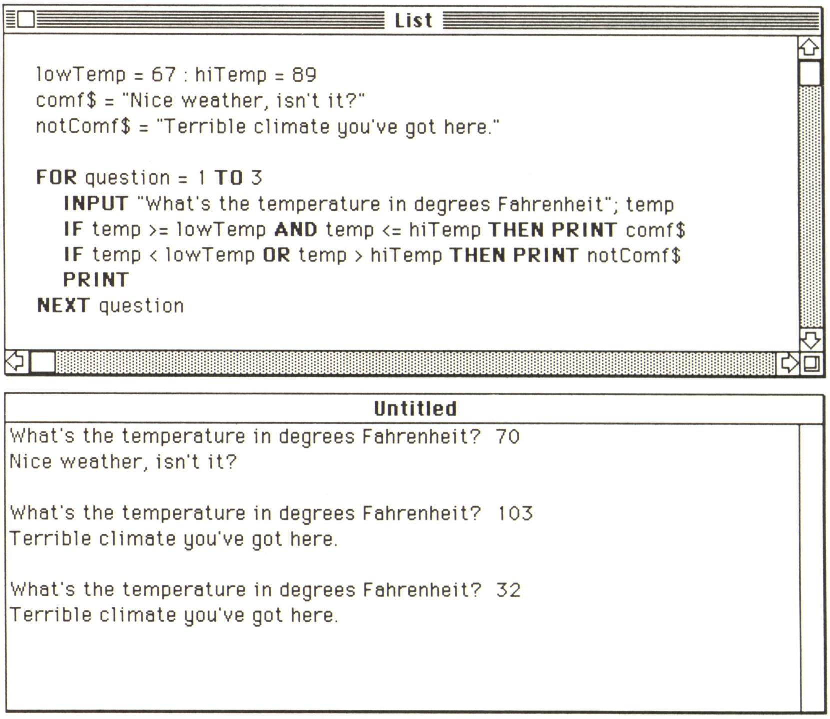
LOGICAL OPERATORS

The power of decision-making statements like IF...THEN can be greatly increased by the use of *logical operators.* You are already familiar with the arithmetic operators +, −, / and *, which perform an operation on two different numbers to yield a third number. Logical operators perform a similar sort of operation on two different logical expressions to yield a logical result, *true* or *false.*

The common logical operators are AND, OR, and NOT. (There are three other logical operators supported by Microsoft BASIC: XOR, IMP, and EQV, but they are less common and we're not going to discuss them in this book.)

Figure 7-6 shows a program that uses the logical operators AND and OR. This program asks us to tell it the temperature. It then makes a remark based on our answer. If the temperature is in the range 67 to 89, the program responds with *Nice weather, isn't it?* If the temperature is lower than 67 or higher than 89, then the response is *Terrible climate you've got here.*

In this program we have again used string variables to shorten the IF...THEN statements. These variables, *comf$* and *notComf$*, are assigned values at the beginning of the program.



```
lowTemp = 67 : hiTemp = 89
comf$ = "Nice weather, isn't it?"
notComf$ = "Terrible climate you've got here."

FOR question = 1 TO 3
   INPUT "What's the temperature in degrees Fahrenheit"; temp
   IF temp >= lowTemp AND temp <= hiTemp THEN PRINT comf$
   IF temp < lowTemp OR temp > hiTemp THEN PRINT notComf$
   PRINT
NEXT question
```

```
What's the temperature in degrees Fahrenheit?  70
Nice weather, isn't it?

What's the temperature in degrees Fahrenheit?  103
Terrible climate you've got here.

What's the temperature in degrees Fahrenheit?  32
Terrible climate you've got here.
```

FIGURE 7-6

Logical operators in the climate program

The AND operator

Notice that to print the first comment in the variable *comf$*, the program must know that the temperature is greater than or equal to *lowTemp*, and also that it is less than or equal to *hiTemp*. That's the purpose of the logical expression in the first IF...THEN statement: *temp >= lowTemp AND temp <= hiTemp*. If this expression is true, then the comment about the nice weather is printed. The AND specifies that the expressions on *both sides* of the AND must be true in order for the entire expression to be true. AND has almost the same function here as it does in everyday English, when you say, "If it is raining, and I drive too fast, then my car will skid off the road."

The OR operator

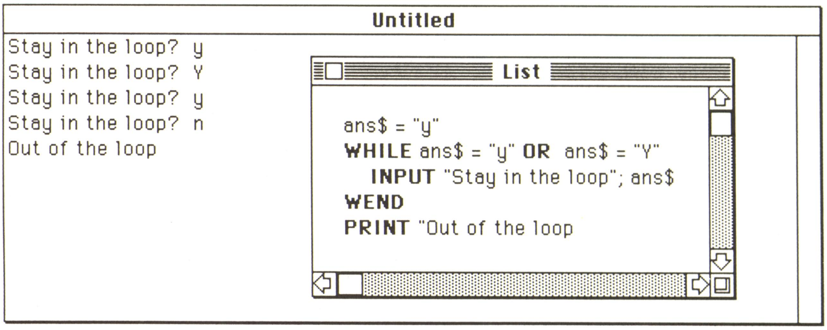
The second IF statement in the program prints the comment about the terrible climate if either the temperature is less than *lowTemp*, OR the temperature is greater than *hiTemp*. Thus if in the logical expression *temp < lowTemp OR temp > hiTemp* the logical expression on *either* side of the OR is true, then the entire logical expression is true. Like AND, the operator OR in BASIC plays a role very similar to the one it plays in everyday English, in such sentences as "If I either die or go mad, then my estate will pass to my heirs."

The interesting thing about these two IF statements is that they check for the same range of temperatures. The only difference is that the first statement needs a number *between* the limits, and the second *outside* the limits. Since these two conditions are the opposite of each other, we could replace them with one IF...THEN...ELSE statement that checks only whether the temperature is inside the comfort range, and then print either one or the other message depending on its truth.

Logical operators in WHILE... WEND loops

Logical operators can also appear in the logical expressions in WHILE... WEND loops, as well as in the logical expressions in IF...THEN statements. Figure 7-7 on the following page shows a short program that uses OR to make itself a little smarter.

Sometimes a program will ask the user to type *y* or *n* in response to a question, and the user will respond in uppercase letters with *Y* or *N*.



```
Stay in the loop? y
Stay in the loop? Y
Stay in the loop? y
Stay in the loop? n
Out of the loop
```

```
ans$ = "y"
WHILE ans$ = "y" OR  ans$ = "Y"
   INPUT "Stay in the loop"; ans$
WEND
PRINT "Out of the loop
```

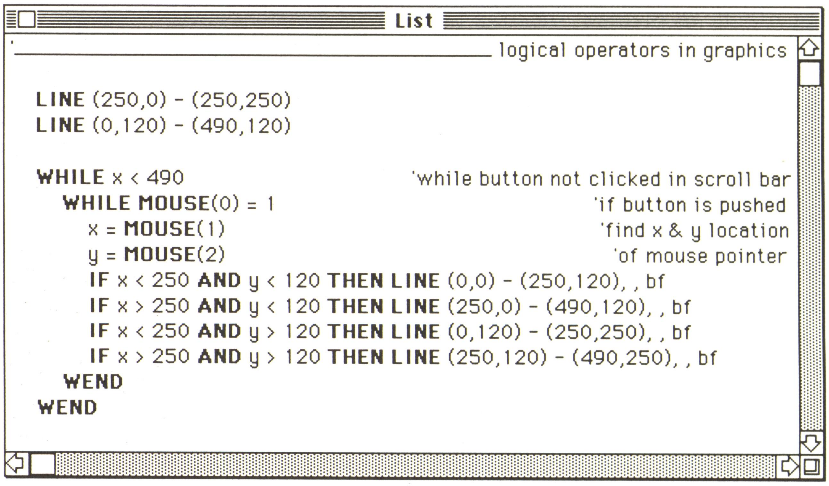
FIGURE 7-7

The logical operator OR in a WHILE… WEND loop

Since this isn't what the program was expecting, it will give erroneous results. The example in Figure 7-7 shows how to broaden the range of acceptable answers to questions by using the OR operator. The logical expression *ans$ = "y" OR ans$ = "Y"* does the trick.

Logical operators in graphics

Logical operators are often necessary when we're writing programs that draw graphics in the Output window. Figure 7-8 shows the listing and sample run for a program that divides the Output window into four quadrants, and then figures out which of the four quadrants the mouse pointer is in when the button is clicked.



```
'___________________________________________________ logical operators in graphics

  LINE (250,0) - (250,250)
  LINE (0,120) - (490,120)

  WHILE x < 490                               'while button not clicked in scroll bar
     WHILE MOUSE(0) = 1                                      'if button is pushed
        x = MOUSE(1)                                         'find x & y location
        y = MOUSE(2)                                           'of mouse pointer
        IF x < 250 AND y < 120 THEN LINE (0,0) - (250,120), , bf
        IF x > 250 AND y < 120 THEN LINE (250,0) - (490,120), , bf
        IF x < 250 AND y > 120 THEN LINE (0,120) - (250,250), , bf
        IF x > 250 AND y > 120 THEN LINE (250,120) - (490,250), , bf
     WEND
  WEND
```

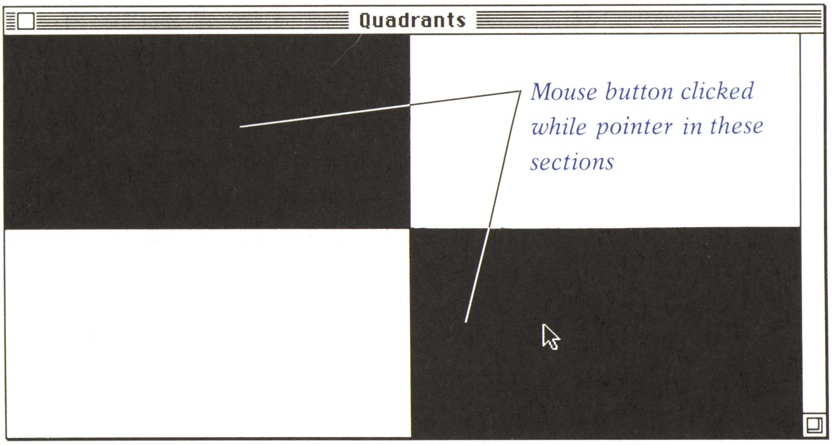
FIGURE 7-8

The listing and a sample run of the quadrants program

(continued)

FIGURE 7-8

(continued)



The program first draws two lines to divide the Output window into four quadrants. The vertical line has a horizontal coordinate of 250, which is about halfway across the window, and the horizontal line has a vertical coordinate of 120, which is about halfway down the window.

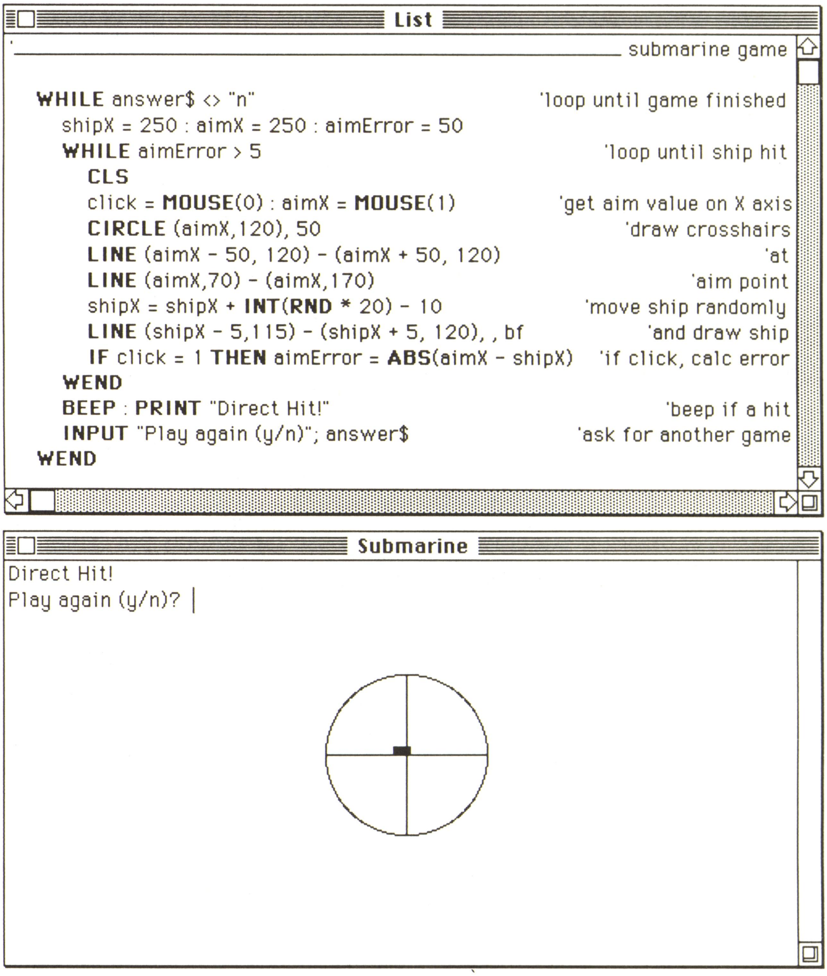
The outer loop of this program keeps the program going as long as the user doesn't click the mouse button in the right scroll bar. The user can do anything with the mouse in the Output window itself, but clicking in the scroll bar, where *x* is greater than 490, causes an exit from the outer WHILE...WEND loop. This offers an easy way to exit from the program.

The program then checks to see if the mouse button has been clicked with the logical expression *MOUSE(0)=1* in the inner WHILE statement. If the mouse button has not been clicked, it skips the inner WHILE...WEND loop entirely.

As soon as the button is clicked, the inner WHILE...WEND loop is executed. In this loop, the program reads the coordinates of the mouse pointer into the variables *x* and *y*. It then executes a series of IF...THEN statements to find out where to draw a black rectangle depending on the location of the mouse pointer. For instance, if the pointer is in the upper left quadrant of the window, *x* must be less than 250 and *y* must be less than 120. This is checked in the first of the IF...THEN statements. If the condition is true, the LINE statement draws a filled box in this quadrant of the screen. Similarly, a mouse click occurring in any of the other quadrants causes the conditions in one of the other IF statements to be met, and a box to be drawn in the corresponding quadrant.

SUBMARINE GAME

We'll wrap up this chapter with a short game program that uses many of the principles we've learned so far. The program draws a set of cross hairs on the screen, reminiscent of a submarine periscope, and then causes a small black rectangle, or "ship," to slide back and forth across the screen. The cross hairs can be moved horizontally with the mouse. If the "ship" is lined up with the vertical cross hair when we push the mouse button, we'll score a hit and the program will beep and print the message *Direct Hit!* in the Output window. Figure 7-9 shows the listing for the program and a sample of its operation.



```
'________________________________________________________ submarine game

  WHILE answer$ <> "n"                                'loop until game finished
    shipX = 250 : aimX = 250 : aimError = 50
    WHILE aimError > 5                                      'loop until ship hit
      CLS
      click = MOUSE(0) : aimX = MOUSE(1)                'get aim value on X axis
      CIRCLE (aimX,120), 50                                   'draw crosshairs
      LINE (aimX - 50, 120) - (aimX + 50, 120)                              'at
      LINE (aimX,70) - (aimX,170)                                     'aim point
      shipX = shipX + INT(RND * 20) - 10                   'move ship randomly
      LINE (shipX - 5,115) - (shipX + 5, 120), , bf             'and draw ship
      IF click = 1 THEN aimError = ABS(aimX - shipX)     'if click, calc error
    WEND
    BEEP : PRINT "Direct Hit!"                                    'beep if a hit
    INPUT "Play again (y/n)"; answer$                   'ask for another game
  WEND
```

```
Direct Hit!
Play again (y/n)? |
```

FIGURE 7-9

The listing and output of the submarine program

The comments included with the listing describe the program's operation. Don't worry if every detail of its operation is not clear. It is included here as much for fun as to teach anything new.

There are three important variables in the program: *aimX*, which is the horizontal coordinate of the mouse and therefore of the cross hairs, which follow the mouse; *shipX*, which is the horizontal coordinate of the ship; and *aimError*, which is the difference between these two quantities and a measure of how close the user has come to hitting the ship. The variable *shipX* is changed continuously by a random amount. If *aimError* is less than 5, the user scores a hit, the inner WHILE...WEND loop terminates, and the program beeps and prints the *Direct Hit!* message.

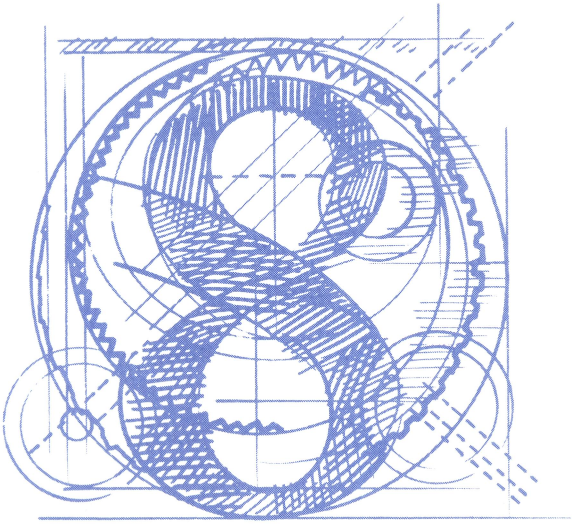
The ABS function

There's one new function in this program: *ABS.* ABS stands for ABSolute value. The absolute value of a number is the number without its sign. Thus ABS(3) is 3, and ABS(−3) is also 3. In our program, we want to calculate *aimError*, which is the difference between where the ship is and where the user is aiming. However, it doesn't matter what the sign of the result is, since aiming too far to the right should be no different than aiming too far to the left. So we calculate *aimError* as the absolute value of the difference between *aimX* and *shipX*.

REFRESH-YOURSELF WINDOW

In this chapter we've learned about the decision-making statements IF... THEN and IF...THEN...ELSE. We've also learned about the logical operators OR and AND, which increase the versatility of the logical expressions in decision-making and other kinds of BASIC statements. Finally, we've covered four new commands: RND, RANDOMIZE, INT, and ABS.

In the next chapter, we will expand our knowledge of BASIC by learning how to organize longer programs with *subroutines.*



Subroutines and Program Organization

So far our program examples have been fairly simple and short—their organization, length, and structure have been such that we could glance at the program and easily get a feeling for what it did. And, except for learning about indenting and adding comments with

REM statements to make our programs more meaningful, we have not really needed to think much about whether a program was understandable or readable to an outsider, or even to *us* later on when we came back to it. If all programs could be this short, there would be no need for further discussion. However, any program worth its salt is usually long and complex. The problem is that as our programs get longer and more complex, it gets increasingly important to maintain a logical order and structure.

Writing a long and complex program is something like building a house. Both processes are overwhelming unless they are reduced to several smaller, more manageable tasks or *modules.* The principal method for structuring complex BASIC programs into smaller modules or tasks is the *subroutine.* This chapter explains how to structure our Microsoft BASIC programs with subroutines. We will also cover the *indexed subroutine.* As a treat you will learn a neat animation game. To wind up the chapter, we will touch lightly on an even more sophisticated method for accessing program modules called *subprograms.*

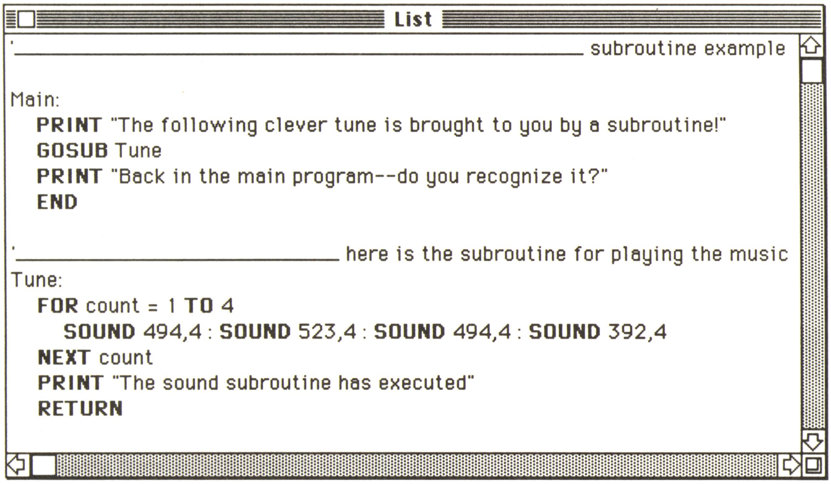
WHAT IS A SUBROUTINE?

A subroutine is a program within a program. It is made up of BASIC statements and performs some simple task. A subroutine is *called* (ordered to execute) by a statement in the *main program.* A subroutine might, for example, PRINT the title of an adventure game, empty the cells of a spreadsheet, compute the odds for a nuclear war, or fill the Macintosh screen with palm trees and turtles from the Cairo font. Continuing our analogy of building a house, programming with subroutines is analogous to using subcontractors—the house is made by subcontractors, under our control, each performing their required tasks in the correct order. The subcontractors do their job when they are *called* on to do so by the *main* builder: us.

Besides allowing programs to be divided into smaller sets of tasks, subroutines are useful when the program needs to perform a series of instructions several times throughout the program. For example, we might have a graphics program that draws four wheels on a truck. Rather than having four individual routines and statements draw the wheels, we could write a single subroutine that does the job of drawing a single wheel. Then each time we needed a wheel, we would simply call the subroutine. To do the whole truck we'd call the subroutine four times, perhaps with a loop.

GOSUB and RETURN

Subroutines are accessed through the GOSUB and RETURN statements. Like GOTO, GOSUB causes the program to branch, or "jump" to a certain



```
'____________________________________________ subroutine example

Main:
  PRINT "The following clever tune is brought to you by a subroutine!"
  GOSUB Tune
  PRINT "Back in the main program--do you recognize it?"
  END

'________________________ here is the subroutine for playing the music
Tune:
  FOR count = 1 TO 4
    SOUND 494,4 : SOUND 523,4 : SOUND 494,4 : SOUND 392,4
  NEXT count
  PRINT "The sound subroutine has executed"
  RETURN
```

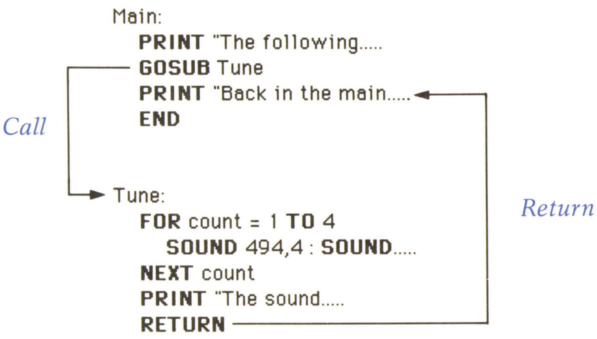
FIGURE 8-1

The GOSUB program

line or label in the program. However, there is a substantial difference between the way GOTO works and the way GOSUB works. See if you can spot the difference in the program in Figure 8-1. Enter it into the List window and run it.

Our program first prints out the message in the first two PRINT statements, then plays a tune from a popular television show. The statement *GOSUB Tune* tells the program to branch to the label called *Tune*. We can say that the subroutine has a *name* and here it's called *Tune*. Next the tune gets played by Macintosh's SOUND statement.

Up to this point GOSUB behaves no differently than GOTO. The difference starts with the RETURN statement at the end of the subroutine. What RETURN does is cause the program to return to the original branch point (the *GOSUB Tune* statement) and then resume execution with the statement immediately following the GOSUB statement. Figure 8-2 illustrates the flow of our program with its subroutine.



```
Main:
  PRINT "The following.....
  GOSUB Tune
  PRINT "Back in the main.....
  END

Tune:
  FOR count = 1 TO 4
    SOUND 494,4 : SOUND.....
  NEXT count
  PRINT "The sound.....
  RETURN
```

FIGURE 8-2

The program flow of the subroutine

So, whereas GOTO causes a one-way branch to the label, GOSUB causes a branch to the label *but also remembers where it came from.* (In Microsoft BASIC, programs may also branch to a subroutine beginning with a line number.) When the RETURN statement is found at the end of the subroutine, the program returns to the branching point and resumes execution at the next statement after the GOSUB it branched from.

GOSUB and RETURN must be used together as a matched pair. The only exception is if you GOSUB to a subroutine that you do not plan to come back from, such as a subroutine that ends the program. But if a RETURN statement is executed accidentally, you may end up back at the statement following the GOSUB that called the subroutine, when you don't want to be there. And if you can't find the random RETURN statement, you will find it hard to understand why your program is not working right. *Dangling RETURNs,* as they are called, are not uncommon in complex BASIC programs. Furthermore, if you GOSUB a great many times without RETURNing, BASIC will eventually be so confused that it will give an "Out of memory" error message.

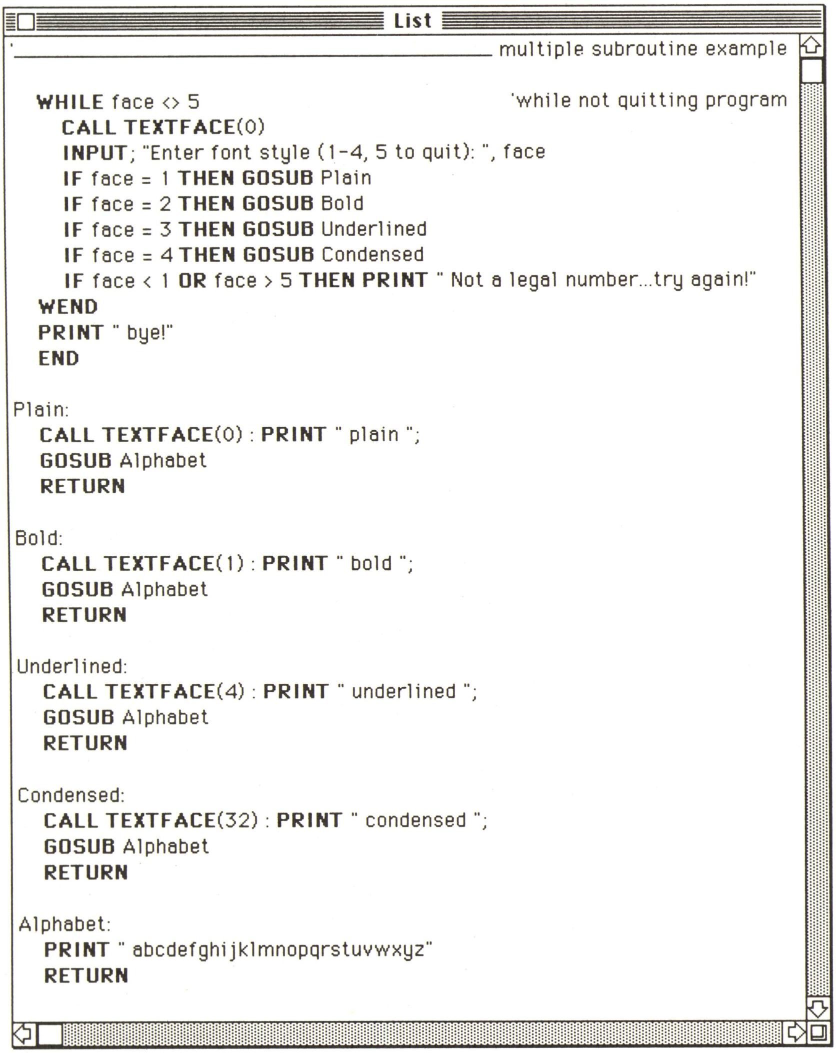
PROGRAM ORGANIZATION WITH GOSUB AND RETURN

Our previous program showed how GOSUB works and how it differs from GOTO. But it did not really show off the usefulness of subroutines. We could have simply placed the *Tune* subroutine code between the two PRINT statements and accomplished the same results. A subroutine's true usefulness becomes apparent when our programs start to get larger and have a more convoluted structure, or when we want to do the same thing over and over from a main program. Let's look at a program that benefits from using subroutines.

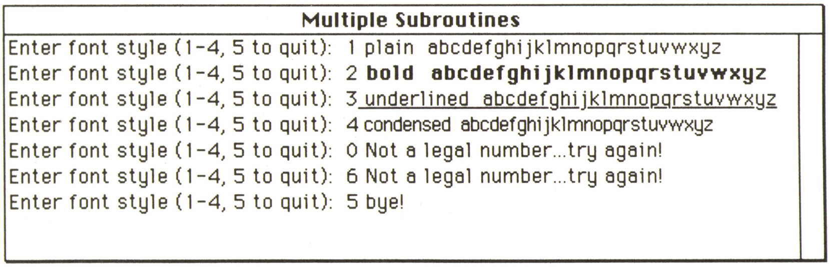
In Chapter 4 we discovered the TEXTFACE statement, which allows us to print text in the Output window with different combinations of text styles and "faces." The program in Figure 8-3 takes off on this statement and assigns the numbers 1 through 4 to the plain, bold, underlined, and condensed textfaces. It then asks the user to enter one of the style numbers. The program looks at the user-input number, branches to the appropriate subroutine, prints a line of text in the appropriate textface, then returns to the main program and asks for another number. If the user types a 5, the program stops. If the user types a number that is not between 1 and 5, the program prints an error message and asks the user to try again.

FIGURE 8-3

The textface program and its output



```
'____________________________________ multiple subroutine example

  WHILE face <> 5                              'while not quitting program
    CALL TEXTFACE(0)
    INPUT; "Enter font style (1-4, 5 to quit): ", face
    IF face = 1 THEN GOSUB Plain
    IF face = 2 THEN GOSUB Bold
    IF face = 3 THEN GOSUB Underlined
    IF face = 4 THEN GOSUB Condensed
    IF face < 1 OR face > 5 THEN PRINT " Not a legal number...try again!"
  WEND
  PRINT " bye!"
  END

Plain:
  CALL TEXTFACE(0) : PRINT " plain ";
  GOSUB Alphabet
  RETURN

Bold:
  CALL TEXTFACE(1) : PRINT " bold ";
  GOSUB Alphabet
  RETURN

Underlined:
  CALL TEXTFACE(4) : PRINT " underlined ";
  GOSUB Alphabet
  RETURN

Condensed:
  CALL TEXTFACE(32) : PRINT " condensed ";
  GOSUB Alphabet
  RETURN

Alphabet:
  PRINT " abcdefghijklmnopqrstuvwxyz"
  RETURN
```



```
Enter font style (1-4, 5 to quit):  1 plain  abcdefghijklmnopqrstuvwxyz
Enter font style (1-4, 5 to quit):  2 bold  abcdefghijklmnopqrstuvwxyz
Enter font style (1-4, 5 to quit):  3 underlined  abcdefghijklmnopqrstuvwxyz
Enter font style (1-4, 5 to quit):  4 condensed  abcdefghijklmnopqrstuvwxyz
Enter font style (1-4, 5 to quit):  0 Not a legal number...try again!
Enter font style (1-4, 5 to quit):  6 Not a legal number...try again!
Enter font style (1-4, 5 to quit):  5 bye!
```

The main part of the program is between the WHILE and WEND statements. Inside this loop the textface is first set to 0, which is the number assigned by BASIC to the plain textface. Everything will be printed in plain text until the next CALL TEXTFACE statement changes the style. Next, the INPUT statement displays the choices and accepts a number from the user to assign to the variable *face*. Note that the INPUT statement is immediately followed by a semicolon. As we learned in Chapter 5, this eliminates the carriage return that normally occurs when we type something into an INPUT statement.

Once the user has entered a number into the variable *face*, the program uses five IF...THEN statements to look at the value of the variable *face* and decide which subroutine to branch to: *Plain, Bold, Underlined,* or *Condensed.* The fifth IF...THEN statement handles the situation when the user of the program types a number not in the required range by printing out the message, *Not a legal number...try again.* The WEND statement sends the program back to the beginning of the WHILE loop.

Let's examine the second subroutine, called *Bold,* which the program branches to if *face* is 2. Here, the statement CALL TEXTFACE(1) selects the bold textface. Next, the word *bold* is printed, and since we have called TEXTFACE(1), the word itself is printed in boldface. Next, there is another subroutine, called *Alphabet.* This is an example of a subroutine calling another subroutine, which we'll discuss later in the chapter.

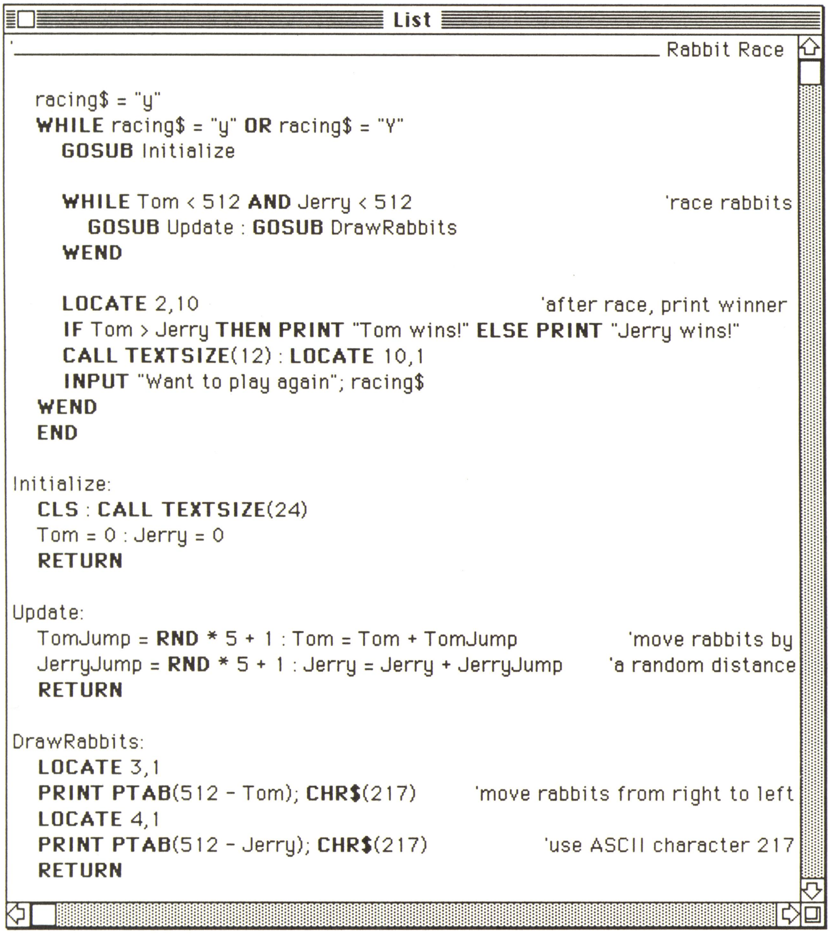
The *Alphabet* subroutine prints the letters *a* through *z* in the textface set by the CALL TEXTFACE statement in the subroutine that called it. After it prints the letters, *Alphabet* returns to the line following the GOSUB statement in the subroutine that called it, which happens to be RETURN in all the textface subroutines. This RETURN sends us back to the main program.

We return to the IF...THEN statement below the one that called the *Bold* subroutine. Because its logical expression is false, the program drops to the next IF...THEN, then the next, and so on, until it finally finishes the WEND and starts the WHILE loop over. Thus only one IF... THEN statement is valid at any one time. When the program starts the loop again, the textface is reset to 0 with the first CALL TEXTFACE statement, guaranteeing that our INPUT statement will print its message in the plain-text style.

Before we leave this program, we should point out that we could have put the PRINT statement that's within the alphabet subroutine within each of the textface subroutines. But that would have meant typing the same line four times. And if we decided later on to add more subroutines to display the other text faces, we would need to type the PRINT statement that many more times. But more important, if we decided later on to change the PRINT statement itself, we would have to change it at every occurrence, instead of just once. These hypothetical circumstances point out just how powerful and useful subroutines can be.

Rabbit Race

Let's look at another GOSUB example, this time in an animated video race between two rabbits, called Tom and Jerry. The program, shown in Figure 8-4, has features of animation, nested loops, random number generation, and many of the BASIC statements we have learned. A sample outcome of the race is shown in Figure 8-5 on the following page.



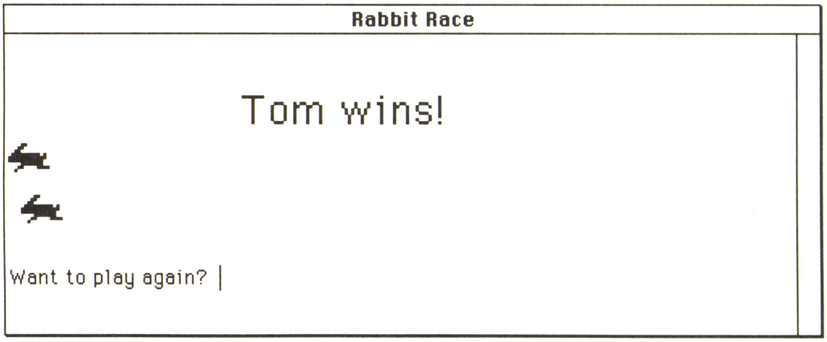
```
'______________________________________________________ Rabbit Race

  racing$ = "y"
  WHILE racing$ = "y" OR racing$ = "Y"
    GOSUB Initialize

    WHILE Tom < 512 AND Jerry < 512                          'race rabbits
      GOSUB Update : GOSUB DrawRabbits
    WEND

    LOCATE 2,10                                     'after race, print winner
    IF Tom > Jerry THEN PRINT "Tom wins!" ELSE PRINT "Jerry wins!"
    CALL TEXTSIZE(12) : LOCATE 10,1
    INPUT "Want to play again"; racing$
  WEND
  END

Initialize:
  CLS : CALL TEXTSIZE(24)
  Tom = 0 : Jerry = 0
  RETURN

Update:
  TomJump = RND * 5 + 1 : Tom = Tom + TomJump               'move rabbits by
  JerryJump = RND * 5 + 1 : Jerry = Jerry + JerryJump     'a random distance
  RETURN

DrawRabbits:
  LOCATE 3,1
  PRINT PTAB(512 - Tom); CHR$(217)          'move rabbits from right to left
  LOCATE 4,1
  PRINT PTAB(512 - Jerry); CHR$(217)                'use ASCII character 217
  RETURN
```

FIGURE 8-4

A program using GOSUB for animated character graphics

FIGURE 8-5

Sample output of the Rabbit Race program



```
                Tom wins!



Want to play again? |
```

In Chapter 4 we learned how to work with graphics using pixels and shapes such as circles and boxes. This program, however, is an example of *character-based graphics* because it uses a *text* character as a *graphic* element. We'll learn more about character-based graphics in Chapter 15.

First, where do the rabbit symbols come from? The rabbit symbol is obtained by using PRINT CHR$(217). We'll learn exactly how BASIC's CHR$ function works in Chapter 10, "Strings." For now you can just think of it as a way to generate special characters, such as the rabbit. The rabbit comes with the Macintosh Geneva font on the Microsoft BASIC disk. We changed the textsize to 24 point to give us a big rabbit.

The program moves the rabbits by keeping track of their horizontal positions on the screen. We use the variables *Tom* and *Jerry* to save these values (*Tom* is the upper rabbit and *Jerry* is the lower). These positions vary from 0 to 512, which is the width of the Macintosh screen in pixels. The program animates the rabbits by using the variables *Tom* and *Jerry* in a PTAB statement. The PTAB causes the text cursor to tab, or move to, the pixel specified by the statement. PTAB ordinarily moves the text cursor from left to right, so if our PTAB statement were to read simply *PRINT PTAB (Tom),* for example, Tom would move from left to right and appear to be hopping backward! We can solve this problem and make Tom and Jerry hop from right to left by subtracting the values of their horizontal positions from 512, which is the location of the right edge of the screen. Understand that we are using PTAB instead of PSET because we are moving *characters,* not single pixels.

The main part of the program is at the beginning, surrounded with a WHILE...WEND loop. There are three subroutines below the main program: *Initialize, Update,* and *DrawRabbits.* Notice that we chose the names of these subroutines to convey what they actually do. Looking at the main program, we see that the program consists of an initialization step, which is followed by a second, nested WHILE...WEND loop that cycles through two steps: *Update* and *DrawRabbits.* Notice how simple the main program looks, thanks to the use of subroutines.

Now that we have an overview of the program and what it does, let's follow BASIC on its merry way through the program.

Beginning the main program

The first line of the program is an assignment statement that assigns the value *y* to the string variable *racing$:*

```
racing$ = "y"
WHILE racing$ = "y" OR racing$ = "Y"
  GOSUB Initialize
```

We need this statement so that the first time through the program, BASIC will run the race without needing the user to tell it to. This will happen because when BASIC comes to the WHILE statement in the next line, the logical expression will be true and BASIC will continue to the next line, *GOSUB Initialize.* This sends the program to the *Initialize* subroutine.

The Initialize *subroutine*

Most programs begin with a subroutine that initializes various parameters for the program. This makes it easier for the programmer to "see" the various defaults. In the *Initialize* subroutine in our program:

```
Initialize:
  CLS : CALL TEXTSIZE(24)
  Tom = 0 : Jerry = 0
  RETURN
```

BASIC first clears the screen with the *CLS* command. Next, *TEXTSIZE(24)* changes the size of any text characters (the rabbits) to 24 point. Then the line *Tom = 0 : Jerry = 0* sets the location for each rabbit to 0, so that each time the race is run both speedsters start the race at the same spot.

Now that the screen is cleared and the rabbits are set to go, the *RETURN* statement sends BASIC back to the statement immediately following the *GOSUB Initialize* statement in the main program.

RETURN*ing to the main program*

BASIC returns to the section of the program that reads:

```
WHILE Tom < 512 AND Jerry < 512
   GOSUB Update : GOSUB DrawRabbits
WEND
```

The statement *WHILE Tom < 512 AND Jerry < 512* tells BASIC that, as long as *both* Tom and Jerry are less than 512 pixels from the right edge of the Output window, it should continue to the next program line, which is *GOSUB Update.*

The Update *subroutine*

In the *Update* subroutine, the variables *TomJump* and *JerryJump* are set to random numbers between 1 and 6:

```
Update:
   TomJump = RND * 5 + 1 : Tom = Tom + TomJump
   JerryJump = RND * 5 + 1 : Jerry = Jerry + JerryJump
   RETURN
```

The statement *Tom = Tom + TomJump* replaces the value of Tom's position with the value of that position plus the random distance held in *TomJump.* So the first time through the loop BASIC could move Tom one pixel, for example, and the next time through the loop five pixels. Obviously, if one rabbit gets larger numbers of TomJump or JerryJump, it will move farther than the other.

After Jerry's location has been updated, the program returns to the next statement in the main program: *GOSUB DrawRabbits.*

The DrawRabbits *subroutine*

The *DrawRabbits* subroutine is where the real work is done:

```
DrawRabbits:
   LOCATE 3,1
   PRINT PTAB(512 - Tom); CHR$(217)
   LOCATE 4,1
   PRINT PTAB(512 - Jerry); CHR$(217)
   RETURN
```

First, *LOCATE 3,1* puts the insertion point at row 3, column 1. The next line, *PRINT PTAB(512 – Tom); CHR$(217),* first subtracts Tom's location from 512 (the right edge of the screen), tabs to the pixel specified by the result of the subtraction, then prints CHR$(217), which is the rabbit symbol. Each time BASIC goes through the main program loop and prints a new image of Tom, the previous image to the right is erased. (We'll explain why this happens in Chapter 17, "Animation," but suffice it to say that there is a block of trailing space at the end of each rabbit character that obliterates the one behind it.)

The program prints Tom, then continues to the next program line, *LOCATE 4,1*, which moves the insertion point to row 4, and repeats the print procedures for Jerry. The end result is that the rabbits are displayed one above the other and race from right to left in their respective rows.

RETURN*ing to the main program*

After the *DrawRabbits* subroutine is completed, the program returns to the statement following the GOSUB statement that sent it to *DrawRabbits,* which is the inner WEND statement. The program then loops to the statement *WHILE Tom < 512 AND Jerry < 512.* Now the program looks at the values of both Tom and Jerry and asks, "Is both the value for Tom *AND* the value for Jerry equal to or greater than 512?" If the answer is "Yes," the program continues to the next program line, which sends it through the *Update* and *DrawRabbits* subroutines again, hence moving both rabbits forward. But if the answer is "No," it means that one of the rabbits has reached the left side of the screen and won the race. This ends the inner WHILE loop, and sends the program to the statements following the inner WEND.

```
    LOCATE 2,10
    IF Tom > Jerry THEN PRINT "Tom wins!" ELSE PRINT "Jerry wins!"
    CALL TEXTSIZE(12) : LOCATE 10,1
    INPUT "Want to play again"; racing$
  WEND
  END
```

The program positions the insertion point for the next PRINT statement, then looks at Tom's location. If it's greater than Jerry's, it means he won the race and *Tom wins!* is printed; if Tom's location is not greater than Jerry's, *Jerry wins!* is printed.

The next line reduces the text size to 12 point and changes the location for printing to row 10, column 1, and the program asks: *Want to play again?* When the user types a response, the program loops to the first WHILE statement. If the user has typed *y* or *Y*, the logical expression is true and the program runs the race again. But if the user has typed any other letter, the logical expression is false and the program ends.

Modifications

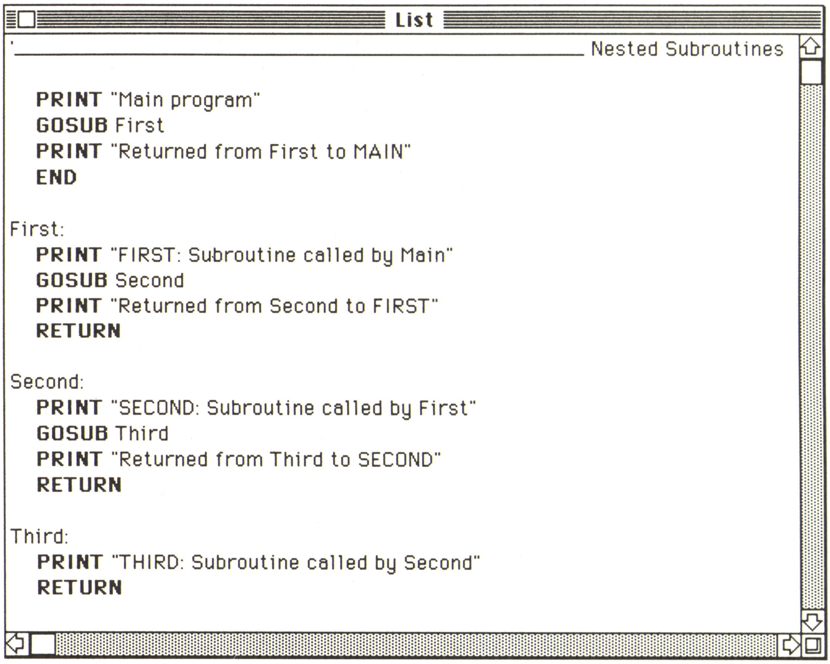
We can make the race visually more interesting by having the faster rabbit appear in an italic face, so that it "looks" faster. We can do this by simply adding two statements to the program.

In the *DrawRabbits* routine, right after the statement *LOCATE 3,1* add the statement: *IF Jerry > Tom THEN CALL TEXTFACE(2) ELSE CALL TEXTFACE(0),* and just after the statement *LOCATE 4,1,* add: *IF Tom > Jerry THEN CALL TEXTFACE(2) ELSE CALL TEXTFACE(0).*

Now that we've created our Rabbit Race program and put Tom and Jerry through their paces, we can see just how useful subroutines are in programs that demand fast looping. That's just one aspect of their power and utility, though, and in the next section we'll discover how to structure our programs with *nested subroutines.*

Nested Subroutines

As we mentioned briefly, we can also have one subroutine call another, in which case we say the second is *nested* within the first. Of course, the subroutine is not really "within" anything; it is the GOSUB statement that is actually nested. Figure 8-6 shows a program that contains three subroutines, the third nested within the second, which is in turn nested within the first. Figure 8-7 shows a flowchart for the program.



```
'_______________________________________________ Nested Subroutines

   PRINT "Main program"
   GOSUB First
   PRINT "Returned from First to MAIN"
   END

First:
   PRINT "FIRST: Subroutine called by Main"
   GOSUB Second
   PRINT "Returned from Second to FIRST"
   RETURN

Second:
   PRINT "SECOND: Subroutine called by First"
   GOSUB Third
   PRINT "Returned from Third to SECOND"
   RETURN

Third:
   PRINT "THIRD: Subroutine called by Second"
   RETURN
```

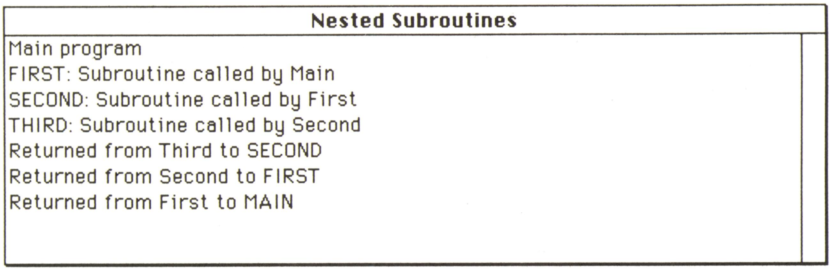
FIGURE 8-6

Three levels of subroutine nesting

(continued)

FIGURE 8-6

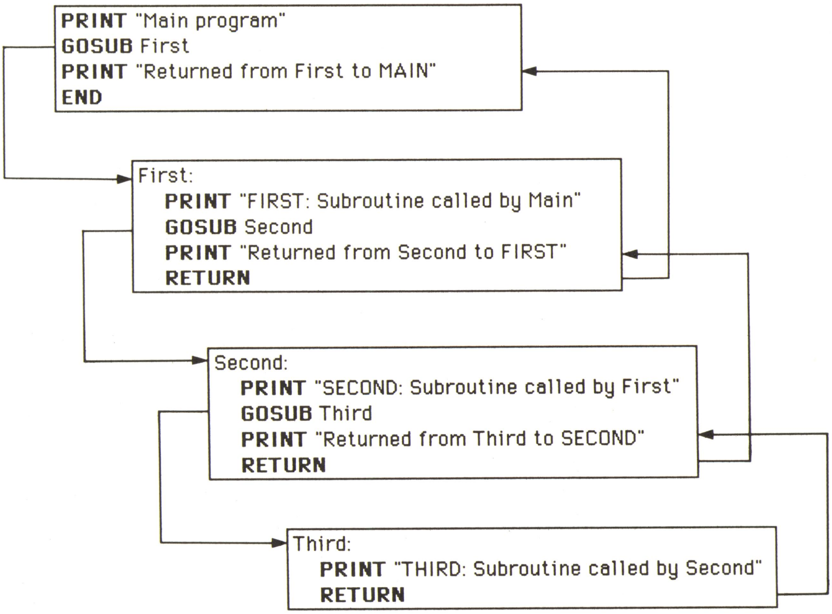
(continued)



```
Main program
FIRST: Subroutine called by Main
SECOND: Subroutine called by First
THIRD: Subroutine called by Second
Returned from Third to SECOND
Returned from Second to FIRST
Returned from First to MAIN
```

FIGURE 8-7

A flowchart for the three levels of subroutine nesting



```
PRINT "Main program"
GOSUB First
PRINT "Returned from First to MAIN"
END

First:
   PRINT "FIRST: Subroutine called by Main"
   GOSUB Second
   PRINT "Returned from Second to FIRST"
   RETURN

Second:
   PRINT "SECOND: Subroutine called by First"
   GOSUB Third
   PRINT "Returned from Third to SECOND"
   RETURN

Third:
   PRINT "THIRD: Subroutine called by Second"
   RETURN
```

The *First* subroutine is called by the main program. It prints out the message, *First: Subroutine called by Main,* then calls the *Second* subroutine. This second subroutine in turn prints a message and calls the *Third* subroutine, which prints its message. The *Third* subroutine then RETURNs to the *Second* subroutine, which then RETURNs to the *First* subroutine, which returns to the main program where the program ends.

Nesting isn't necessarily something that is planned ahead of time. Rather, we usually find that as the structure of a program forms, a routine that is getting long and complex can be decomposed into several subroutines, some of which in turn may be decomposed further.

ON GOSUB: Indexed branching

The ON GOSUB statement is much like the GOSUB statement in that they both call subroutines. However, whereas GOSUB calls one subroutine and returns from it, ON GOSUB chooses one of any number of specified subroutines based on the value of an index. So we use ON GOSUB whenever a program needs to branch to just one of several possible subroutines. Until now, we would have needed to use IF...THEN statements to decide which of several subroutines to go to. Our textface program in Figure 8-3 uses several IF...THEN statements to select just one of four subroutines to execute based on the user's selection. Can we use ON GOSUB instead?

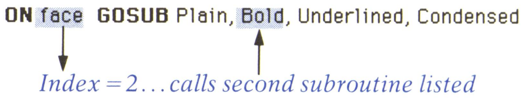
Sure can. Look at Figure 8-8. Here is our textface program redone using ON GOSUB. All that has changed is the IF...THEN statements, which have been replaced with this ON GOSUB statement:

```
ON face GOSUB Plain, Bold, Underlined, Condensed
```

When we run the program, it works exactly like the version that used IF...THEN statements. Let's follow the program through its paces.

After the INPUT statement accepts a number from the user, the program comes to the ON GOSUB statement. If *face* is 1, then the program branches to the first subroutine in the ON GOSUB—in this case subroutine *Plain*. The variable *face* is called the *index* of the ON GOSUB. If this index equals 2, the program then branches to the second subroutine in the series, and if the index equals 3, the program jumps to the third subroutine, and so on. Note that the order of the *subroutine names* in the ON GOSUB is the key to which one will be executed for a given value of the index, not the order of the subroutines in the listing.

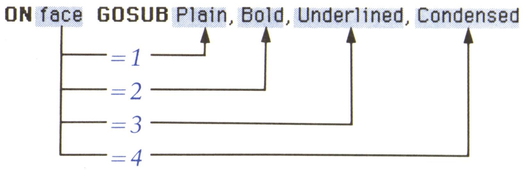
Here is a diagram that summarizes how ON GOSUB makes its selection. We assume the user typed 2, so the index value is 2.



```
ON face GOSUB Plain, Bold, Underlined, Condensed
```

Index = 2... calls second subroutine listed

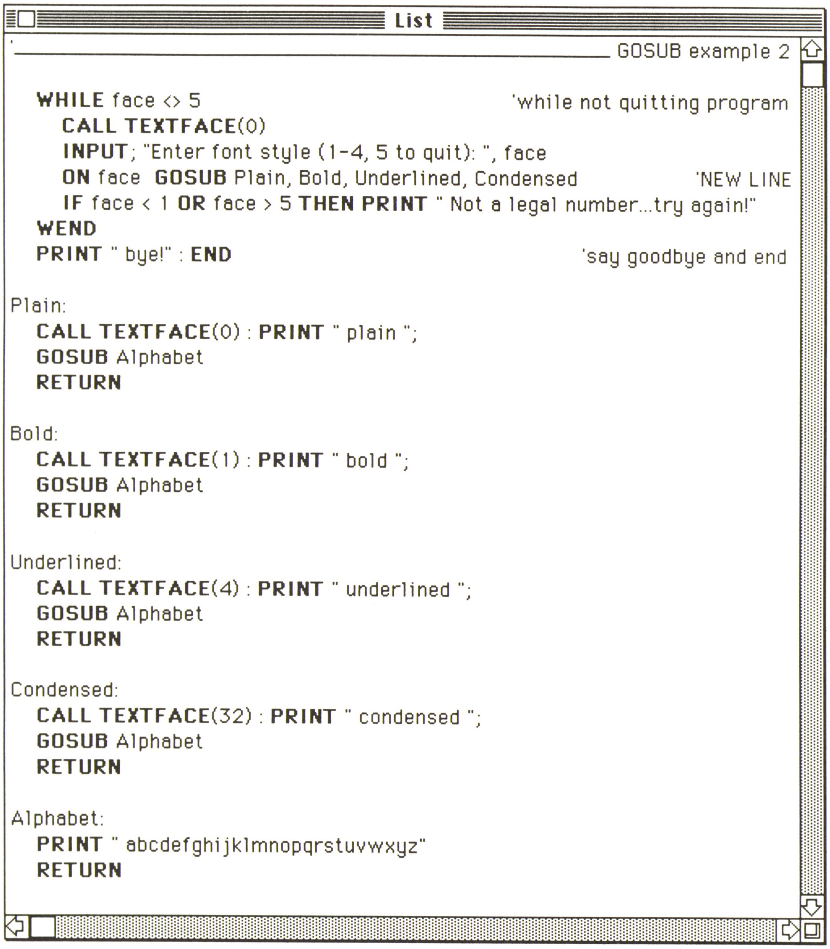
The way ON GOSUB works for other index values is shown here:



```
ON face GOSUB Plain, Bold, Underlined, Condensed
```

FIGURE 8-8

The textface program revised to use ON GOSUB



```
'_______________________________________________________ GOSUB example 2

  WHILE face <> 5                                    'while not quitting program
    CALL TEXTFACE(0)
    INPUT; "Enter font style (1-4, 5 to quit): ", face
    ON face  GOSUB Plain, Bold, Underlined, Condensed              'NEW LINE
    IF face < 1 OR face > 5 THEN PRINT " Not a legal number...try again!"
  WEND
  PRINT " bye!" : END                                      'say goodbye and end

Plain:
  CALL TEXTFACE(0) : PRINT " plain ";
  GOSUB Alphabet
  RETURN

Bold:
  CALL TEXTFACE(1) : PRINT " bold ";
  GOSUB Alphabet
  RETURN

Underlined:
  CALL TEXTFACE(4) : PRINT " underlined ";
  GOSUB Alphabet
  RETURN

Condensed:
  CALL TEXTFACE(32) : PRINT " condensed ";
  GOSUB Alphabet
  RETURN

Alphabet:
  PRINT " abcdefghijklmnopqrstuvwxyz"
  RETURN
```

So, in our program the value of the index, *face*, is used to branch to one of the four subroutines. In essence, the ON GOSUB and the index do the decision making normally done by IF...THEN statements. But using ON GOSUB is much simpler and takes up less room.

The number of subroutines we can use in an ON GOSUB is limited by the maximum value of the index. In Microsoft BASIC, the index can be as large as 255, meaning there can be up to 255 items in the GOSUB list. It is unlikely you'd want so many subroutines in one ON GOSUB: It would be very difficult to edit them, not to mention the problems of printing out the line or explaining to someone else how it worked.

There are a few other limits to the value for the index. If the index is zero or greater than the number of items in the list, but less than or equal to 255, BASIC continues with the next executable statement following the ON GOSUB. If the value of the index is negative or greater than 255, an "Illegal function call" error message is produced.

This points out a flaw in our using an ON GOSUB statement as a replacement for a list of IF...THENs, as we have done in our example. Earlier when we used IF...THENs, the value of *face* could be anything, and it would not cause an error that would stop the program. The last IF...THEN caught the offending value and sent us back for a repeat, informing us of the mistake. But look what happens if we type in too large (greater than 255) or too small a number (negative) in our ON GOSUB version: The ON GOSUB sees the value but can't act on it, causing an "Illegal function call" error message and stopping our program.

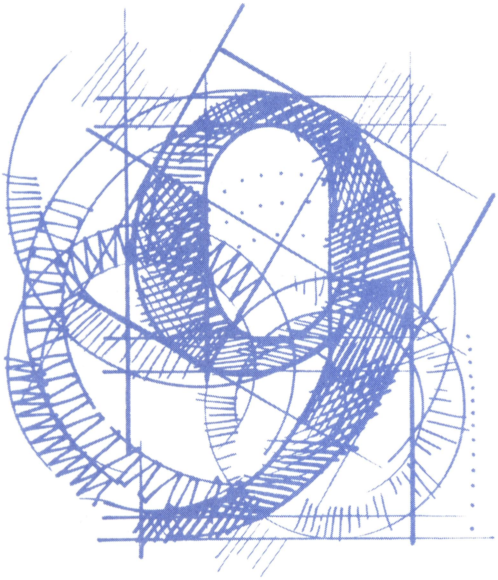
What can be done? In order to remedy the situation in Microsoft BASIC, we would need to include a "trap" for values of *face* that are too big or too small *before* the ON GOSUB statement. We need to "protect" ON GOSUB from offending values. It is not that this is difficult to do, only that it adds a bit to the complexity of our programs. But despite this qualification, ON GOSUB lends itself to simplifying our programs significantly.

BASIC SUBPROGRAMS

There is one other class of Microsoft BASIC program statements that allow better organization and structuring. The *subprogram statements,* consisting of SUB, END SUB, EXIT SUB, and SHARED, were added to Microsoft BASIC for the Macintosh to make our programming more powerful. We won't go into detail about these because they are quite advanced. They are especially useful for creating libraries of routines that can be merged into our BASIC programs without conflicts. For example, a good use for a subprogram would be to create an additional class of graphics statements, perhaps turtle graphics. Just keep in mind that as difficult as they are to understand, these subprograms allow us to create subroutines that can be used in various programs *without having to rewrite all the variable names.*

REFRESH-YOURSELF WINDOW

We have just learned how to use powerful subroutines, BASIC statements for controlling the organization and efficient operation of a program. The combination of GOSUB and RETURN statements allows us to call a subroutine that will branch back to the main program. We have seen how subroutines let us use a section of the program over and over. And ON GOSUB gives the added flexibility of having the program choose a particular subroutine according to the value of an index variable. In the next chapter we will learn how BASIC manipulates lists of data in *arrays.*



Arrays and Read-Data

Programming, as you have probably guessed, is frequently concerned with the efficient organization of information and data. Programs can be written to print out lists of names, assign values to groups of variables, compare different values, analyze codes, determine formats, and so on. The success and performance of these programs rest heavily on how the information is organized. So far we have seen that we can represent information and data very nicely by using string and numeric variables. In this chapter we are going to learn about a variable that will bring a higher level of structure to our programs: the *array variable.* An array variable lets us store different values under the same name. The array itself is a BASIC structure we will use over and over.

We will also learn about the DATA, READ, and RESTORE statements. These statements allow us to store information and data in program lines. Besides storing the data in a way that is highly visible, these statements let us assign fixed data to variables just when we want to.

INSIDE THE ARRAY

A variable is a way of storing a piece of information in a computer program so that it can be used and manipulated. We have seen that both numeric and string variables are represented in a program by using a distinct variable name. Variables that hold only one value at a time are referred to as *simple variables.* For example, the variable *word$* can store only one string at a time. It can contain any string we want, from "BigWig" to "Seattle, Washington", but it can contain only one string at a time. On the other hand, an *array variable* can contain several different values at the same time. It is like a group of variables that all have the same name. Each value of the array is accessed with a *numeric subscript,* which appears in parentheses following the array-variable name. The individual units of the array, which correspond to simple variables, are called the *array elements.* Figure 9-1 shows how an array is organized.

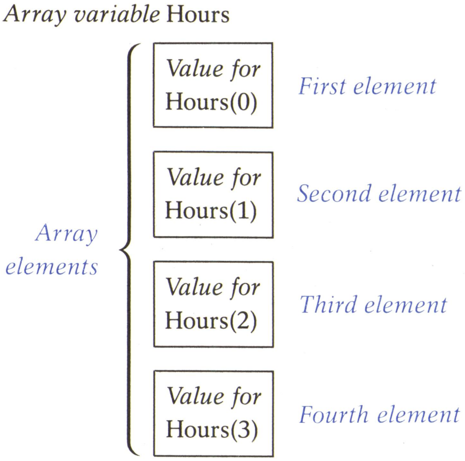


FIGURE 9-1

The organization and structure of arrays

The array variable shown in Figure 9-1 is named *Hours.* We can call an array variable any name we want, as long as it does not conflict with the rules of variable names we learned in Chapter 5.

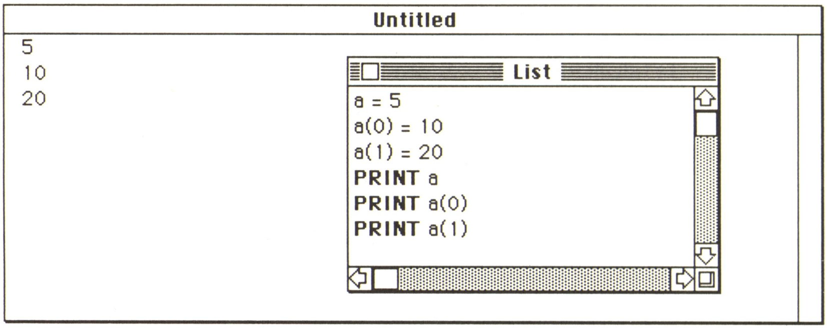
The size limits of an array are set by the Microsoft BASIC interpreter. *Hours* here contains four elements: Each element of an array is identified by its numeric subscript, the number that appears inside the parentheses after the array name. In our example, the first element of the array is called *Hours(0),* the second element is *Hours(1),* the third *Hours(2),* and the fourth *Hours(3).*

It may seem strange to start the first element with the number 0, since it makes the element number different from the number inside the parentheses. It's this way because computers count funny, starting at 0 rather than from 1. But it's a free country and we don't have to follow the computer's numbering technique if we don't like it. We could just as well call the first element *Hours(1),* the second *Hours(2),* and so on, and BASIC would never complain. From now on we may do that, and although the (0) element will still be available, we just won't always choose to use it.

Each of the elements of an array can hold a *numeric value,* indicated by the boxes in the figure. Thus we can assign four unique values to the elements of the array variable *Hours.* We can change the value of a specific element of an array any time we want. We assign numbers to an element of the array just like we do simple variables, except the subscript number is necessary to identify the specific element of the array we wish to assign the number to. For example, if we type:

```
Hours(1) = 44
```

the contents of element 1 of the the array variable *Hours* will contain the value *44.* If we follow this with *Hours(1) =22,* the variable will change to the value 22. We can assign the value 25 to the element *Hours(2),* or 10000 to *Hours(3),* or even −200 to *Hours(1).*



```
5
10
20
```

```
a = 5
a(0) = 10
a(1) = 20
PRINT a
PRINT a(0)
PRINT a(1)
```

FIGURE 9-2

Simple versus array variables

Let's create a program that compares and contrasts simple and array variables. The program and its results are shown in Figure 9-2. We see that the variable *a, a(0),* and *a(1)* are all distinct variables and don't interfere with each other. In this example, *a* is a simple variable, while *a(0)* and *a(1)* are array variables.

THE DIM STATEMENT: SETTING UP ARRAYS

Like simple variables, array variables store their values in the memory space of the computer. With a simple variable, BASIC knows how much memory space it needs because each variable always takes the same amount. But with an array variable, the size of the array depends on how many elements are in the array, and this figure must ultimately be decided by the master programmer, who by now should be you.

Microsoft BASIC provides a statement called *DIM* that sets up the size of an array. DIM stands for *dimension.* The statement:

```
DIM Hours(4)
```

sets up space for an array, *Hours,* with five elements (counting the 0th element). This statement must appear at the beginning of a program, or in such a way that it is executed *before* the array elements are accessed. When the DIM statement is first encountered, the values in its elements are set to zero. To set up an array with 4,001 elements (counting the 0th), we would type:

```
DIM Shoes(4000)
```

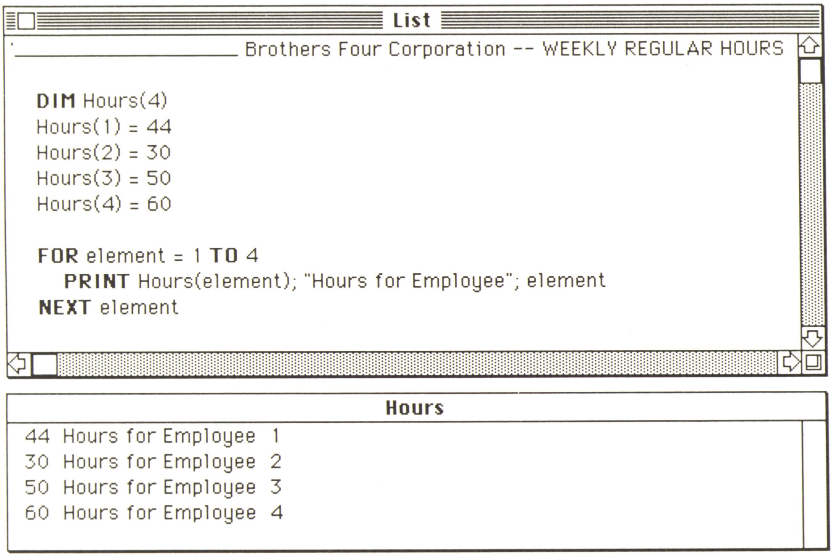
When the DIM statement is optional

Microsoft BASIC has a special feature for setting up arrays. If we use an array with 10 or fewer elements, meaning that we don't try to access element number (11) or above, we do not have to use the DIM statement at the beginning of the program. What happens is that BASIC automatically sets aside 11 elements (0 through 10) for the array when we type a statement using it. Of course, there is nothing to prevent us from using DIM statements with arrays that have fewer than 11 elements, and a statement like *DIM Hours(4)* is perfectly legal. In fact for clarity, it is helpful to include the DIM statement at all times because it shows the reader right away what size array the program needs.

REFERENCING ARRAYS WITH VARIABLE SUBSCRIPTS

Now let's create a program that uses array variables to do some useful work for us. The program and its output are shown in Figure 9-3.

The program uses an array variable to hold the hours that four employees of the Brothers Four Corporation worked for a week. The DIM statement specifies that the array variable *Hours* contains five elements (although we're only using four). Next, the four assignment statements put values in four of the array elements. The array element *Hours(1)* contains the 44 hours that employee 1 worked, the array element *Hours(2)* contains the 30 hours that employee 2 worked, and so on. Thus these element numbers correspond to employees.



```
'__________________ Brothers Four Corporation -- WEEKLY REGULAR HOURS

  DIM Hours(4)
  Hours(1) = 44
  Hours(2) = 30
  Hours(3) = 50
  Hours(4) = 60

  FOR element = 1 TO 4
     PRINT Hours(element); "Hours for Employee"; element
  NEXT element
```

```
 44  Hours for Employee  1
 30  Hours for Employee  2
 50  Hours for Employee  3
 60  Hours for Employee  4
```

FIGURE 9-3

A program using array variables

The FOR...NEXT loop that follows the assigning of the array elements prints out the contents of the array. The PRINT statement prints out the four values assigned to the elements found in the array-variable *Hours(element),* with the elements going from 1 to 4. Note how simple it is to write a loop that examines the elements of an array. If we were using individual simple variables, it would take four separate variables for employee hours and four PRINT statements to display the same information.

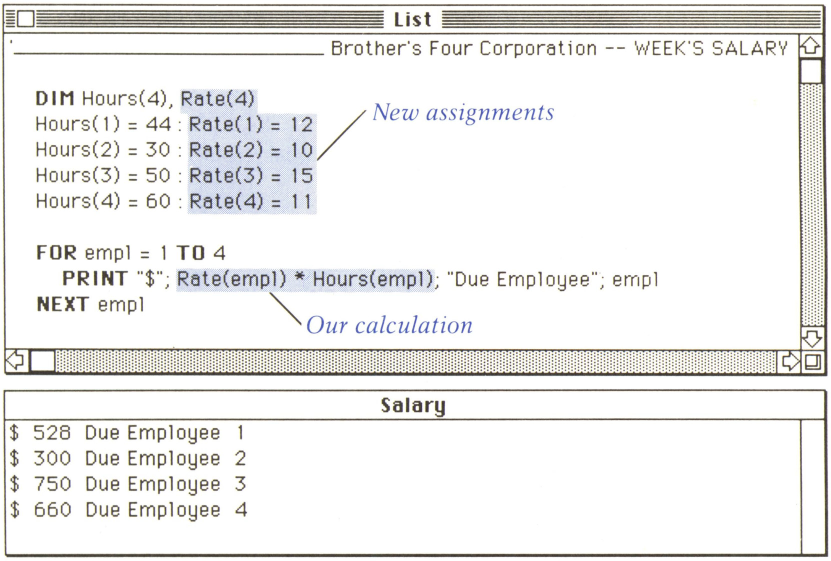
What makes arrays so flexible is that instead of manipulating each variable in a collection of data, we can manipulate the information collectively. We *index* into a collection of organized information. This indexing provides a new degree of freedom. In our corporate-hours example, each employee is specified "symbolically" by an element number. Element 1 is employee 1, element 2 is employee 2, and so on. When we discuss element(2), we refer to the second employee. In fact, we might as well call the index of the FOR...NEXT loop something like *empl* for *employee.*

ARRAY ARITHMETIC

Just as we can perform arithmetic on simple variables, array variables can be added together, multiplied, divided, and subtracted. In fact, all the numeric functions of BASIC can operate on array variables. Calculations and assignments can be made between variables within the same array or between variables of different arrays, or even between arrays and simple variables. The following calculations are all legal uses of BASIC arithmetic on array variables:

```
Hours(5) = Hours(1) + Hours(2)
D(1) = A(5) * B(2) / 2
MagicNumber(3145) = RND * 5 + 20
```

Now let's see how array arithmetic can make life easier for the Brothers Four Corporation by expanding the program so that we can compute the employees' weekly salaries. Since these employees get paid by the hour, and we know how many hours each employee worked in a week, all we need is the *rate of pay* for each employee to figure their salary, because *salary = pay rate per hour × hours worked.* Each employee at the Brothers Four Corporation is paid a different rate, and therefore we need some way to input these rates into the program. Further, we know that each rate must be multiplied by the hours worked for each employee to get his or her



```
'_______________________________ Brother's Four Corporation -- WEEK'S SALARY

DIM Hours(4), Rate(4)
Hours(1) = 44 : Rate(1) = 12
Hours(2) = 30 : Rate(2) = 10
Hours(3) = 50 : Rate(3) = 15
Hours(4) = 60 : Rate(4) = 11

FOR empl = 1 TO 4
  PRINT "$"; Rate(empl) * Hours(empl); "Due Employee"; empl
NEXT empl
```

New assignments

Our calculation

```
$ 528  Due Employee  1
$ 300  Due Employee  2
$ 750  Due Employee  3
$ 660  Due Employee  4
```

FIGURE 9-4

Two arrays used in an arithmetic formula

salary. So what we need is a *second* array to hold the pay rate for each employee, and then a formula to multiply each element of each array together, as shown in Figure 9-4.

In addition to the old *Hours* array, our expanded program contains a new array in the DIM statement called *Rate,* and the (4) tells us that it has five possible elements. Although we could have used a separate DIM statement for *Rate,* BASIC allows us to put multiple array names in a single DIM statement as long as they are separated by commas. *Rate(4)* contains five elements, and four of the elements each hold the rate of pay for one of the four employees.

Following the DIM statement are assignment statements that set up the hours worked and the rate for each employee. Note that the lines have been carefully arranged so that the first line contains both the hours and rate for the first employee, the second line both the hours and rate for the second employee, and so on. We didn't need to do it this way, because BASIC would accept them in any order. But by arranging them as shown we can quickly see how the program is structured. Structuring such a simple program may not seem important now, but when programs get large, they will become unwieldy unless this kind of formatting is done.

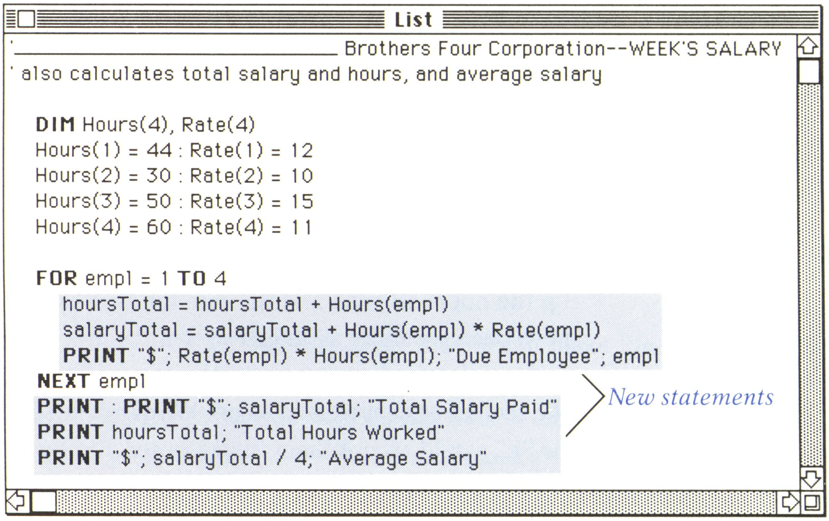
Now that all the information is properly stored in the array, how do we do the computations on the elements of the array? Here is where the beauty of the organized nature of the array shows. All we need to do is make some very simple modifications to our PRINT statement in the loop.

Previously, this statement printed out just the hours worked for each employee. We have replaced its contents completely, adding a leading dollar sign, followed by an arithmetic expression:

```
Rate(empl) * Hours(empl)
```

Recall that *empl* corresponds to the employee number, and can be between 1 and 4. This expression multiplies the hourly rate by the hours worked and produces the weekly salary. When *empl* is 1, we access element 1 for each of the arrays and thus the calculated results belong to employee 1 (think how complex it would be to do this calculation using simple variables). Following the array calculation in the PRINT statement is the string *Due Employee,* and this in turn is followed by the *empl* variable. The result is the four nicely formatted lines shown in the Output window (we left out such niceties as tabbing and pixel tabbing, but you could easily add them).

Our program would be even more useful and informative if it displayed the total hours worked by all employees for the week, the total salaries paid out, and the average salary paid out. Figure 9-5 shows how the program can be expanded to accomplish this.



```
'_______________________________ Brothers Four Corporation--WEEK'S SALARY
' also calculates total salary and hours, and average salary

  DIM Hours(4), Rate(4)
  Hours(1) = 44 : Rate(1) = 12
  Hours(2) = 30 : Rate(2) = 10
  Hours(3) = 50 : Rate(3) = 15
  Hours(4) = 60 : Rate(4) = 11

  FOR empl = 1 TO 4
    hoursTotal = hoursTotal + Hours(empl)
    salaryTotal = salaryTotal + Hours(empl) * Rate(empl)
    PRINT "$"; Rate(empl) * Hours(empl); "Due Employee"; empl
  NEXT empl
  PRINT : PRINT "$"; salaryTotal; "Total Salary Paid"
  PRINT hoursTotal; "Total Hours Worked"
  PRINT "$"; salaryTotal / 4; "Average Salary"
```

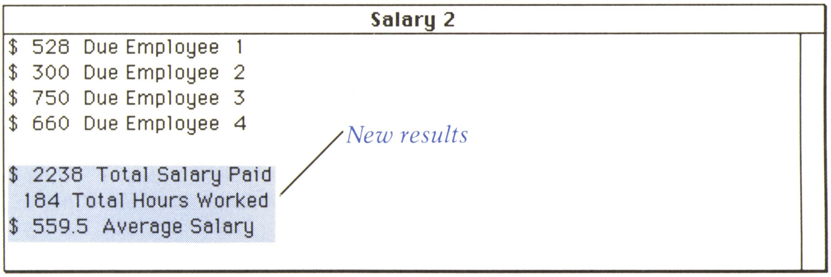
FIGURE 9-5

The expanded salary program

(continued)

FIGURE 9-5

(continued)



```
$ 528 Due Employee 1
$ 300 Due Employee 2
$ 750 Due Employee 3
$ 660 Due Employee 4

$ 2238 Total Salary Paid
  184 Total Hours Worked
$ 559.5 Average Salary
```

New results

ARRAYS AND STRINGS

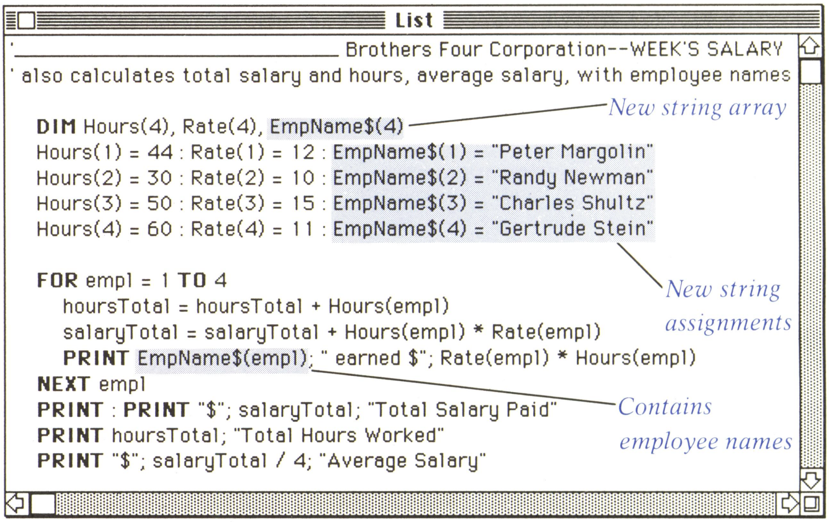
So far we have used only numbers with our array variables. But just as there are two kinds of simple variables, number and string, array variables can also be number and string types. And, like simple string variables, string-array variables can hold strings but not numbers. A string-array variable is identified by a dollar sign following its name, just as simple variables are. These are elements of legal string-array variables:

```
A$(12)
OneWayTicket$(0)
Walla.Walla.Washington$(567)
LotsaLuck$(23)
```

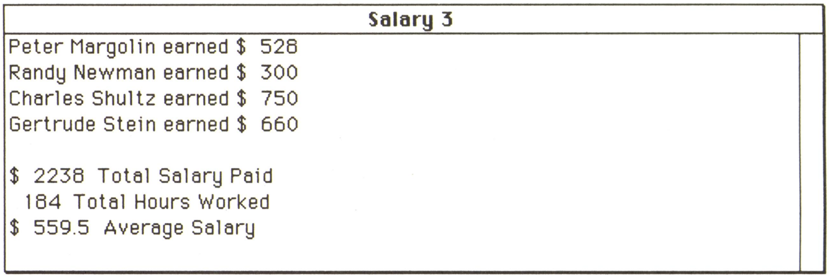
String-array variables are also dimensioned with the DIM statement according to the same rules we followed with number-array variables. Here are some legal DIM statements for the preceding strings:

```
DIM A$(50), OneWayTicket$(10)
DIM Walla.Walla.Washington$(800)
DIM LotsaLuck$(24)
```

Why would we want to use strings with arrays? Let's take a look at our salary program again. Perhaps one day in the future people will have names like *Employee(2),* or maybe even *Person(0).* But we want our programs to have more of a flavor of humanity and friendliness. One way we could humanize the program is to integrate the names of the Brothers Four Corporation's employees into it. which means that we need to add a *third* array. But this time we need a string-array variable, because the names are strings. Figure 9-6 on the following page displays the expanded program and new output.



```
'____________________________ Brothers Four Corporation--WEEK'S SALARY
' also calculates total salary and hours, average salary, with employee names

  DIM Hours(4), Rate(4), EmpName$(4)
  Hours(1) = 44 : Rate(1) = 12 : EmpName$(1) = "Peter Margolin"
  Hours(2) = 30 : Rate(2) = 10 : EmpName$(2) = "Randy Newman"
  Hours(3) = 50 : Rate(3) = 15 : EmpName$(3) = "Charles Shultz"
  Hours(4) = 60 : Rate(4) = 11 : EmpName$(4) = "Gertrude Stein"

  FOR empl = 1 TO 4
    hoursTotal = hoursTotal + Hours(empl)
    salaryTotal = salaryTotal + Hours(empl) * Rate(empl)
    PRINT EmpName$(empl); " earned $"; Rate(empl) * Hours(empl)
  NEXT empl
  PRINT : PRINT "$"; salaryTotal; "Total Salary Paid"
  PRINT hoursTotal; "Total Hours Worked"
  PRINT "$"; salaryTotal / 4; "Average Salary"
```



```
Peter Margolin earned $  528
Randy Newman earned $  300
Charles Shultz earned $  750
Gertrude Stein earned $  660

$  2238  Total Salary Paid
  184  Total Hours Worked
$  559.5  Average Salary
```

FIGURE 9-6

String-array and string-assignment statements added to the salary program

Our program is expanding! But it's not really getting much longer or more complicated. Our DIM statement reveals that we've added a new string array called *EmpName$(4)* with five elements. A new group of assignment statements puts the employees' names into four of the *EmpName$* array elements. Our loop PRINT statement has been changed to print the name of the employee, followed by the salary calculation.

MULTI-DIMENSIONAL ARRAYS

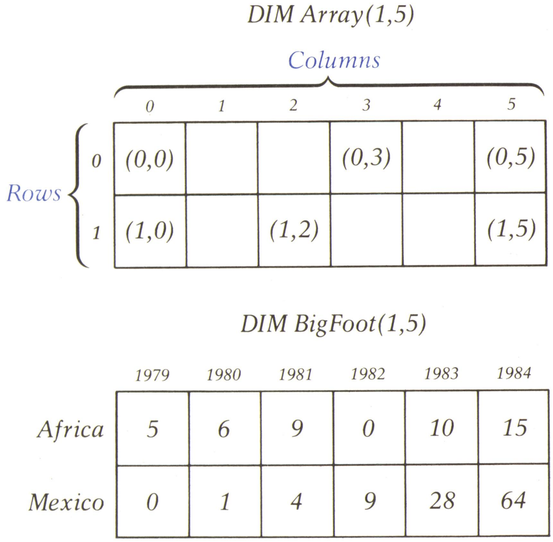
Information is not always collected in the simple formats we've used so far, where one array variable holds only one category of elements. Instead, different elements of a subject are often grouped under a main-category variable. For example, suppose we wanted to store rainfall statistics for a city for each month of the year. This would be an array of 12 elements. Now, if we wanted to do the same for 100 cities, we would need a way of repeating this 12-element array 100 times. *Multidimensional arrays* are BASIC's way of making such organization possible in our programs.

What a multidimensional array does is organize multiple groups of data under one group-variable name. A multidimensional array is set up by using numbers separated by commas inside the parentheses that follow the array name in a DIM statement. Each number represents the size of one *dimension* of the array. The size of a dimension is the number of elements for the dimension. We can think of a two-dimensional array in terms of rows and columns to make it easier.

Figure 9-7 illustrates the concept of two-dimensional arrays; it shows an example that represents the sightings of Bigfoot in Africa and Mexico during the years 1979 through 1984. The columns represent the years. The rows represent the countries: Africa is row 0, and Mexico is row 1. The contents of each element in the array represent the number of sightings for a particular year in a particular country. The two-dimensional array in this example would be defined with the statement:

```
DIM BigFoot(1,5)
```

The two numbers in parentheses specify the two dimensions. The first value in the DIM statement, 1, indicates that we have two rows in the array, since 0 is a legal element. Thus we have row *BigFoot(0,year)* and row *BigFoot(1,year)*. The second value in the DIM statement, 5, indicates that we have six elements in each of the two rows, or six columns. Thus the element *BigFoot(0,3)* is different from the element *BigFoot(1,3)*. When we



DIM Array(1,5)

	0	1	2	3	4	5
0	(0,0)			(0,3)		(0,5)
1	(1,0)		(1,2)			(1,5)

DIM BigFoot(1,5)

	1979	1980	1981	1982	1983	1984
Africa	5	6	9	0	10	15
Mexico	0	1	4	9	28	64

FIGURE 9-7

The concept of two-dimensional arrays, with an example say *BigFoot(0,3)* we are referring to row 0, column 3. *BigFoot(1,5)* would be row 1, column 5.

To store or access information in a multidimensional array we use the same arrangement that appears in the DIM statement. For example, *BigFoot(1,5) = 64* sets sightings for the country Mexico in the year 1984 to the value 64. Likewise *PRINT BigFoot(0,1)* prints out the number of sightings for Africa in 1979.

Although this diagram works well for simple two-dimensional arrays, we can have arrays in BASIC that have three, four, and more dimensions. Although working with them is beyond the scope of this book, arrays of several dimensions find practical use in science and engineering.

Multidimensional arrays are complicated but very useful. A good example of a two-dimensional array is a football-pool card. Recall how a football pool works. You choose two numbers for the pool. The numbers represent what you believe the last digits of the final score for the game will be. For example, if you pick the numbers (2,7) and we assume the visitors' score comes first and the home team's second, you are saying that the final score will be such that the last digit for the visitors will be 2 and for the home team 7. The scores 22 to 37, 12 to 7, or 122 to 177 all qualify you as the winner of the pool. The pool is represented on a card that has 100 cells, like the one shown in Figure 9-8.

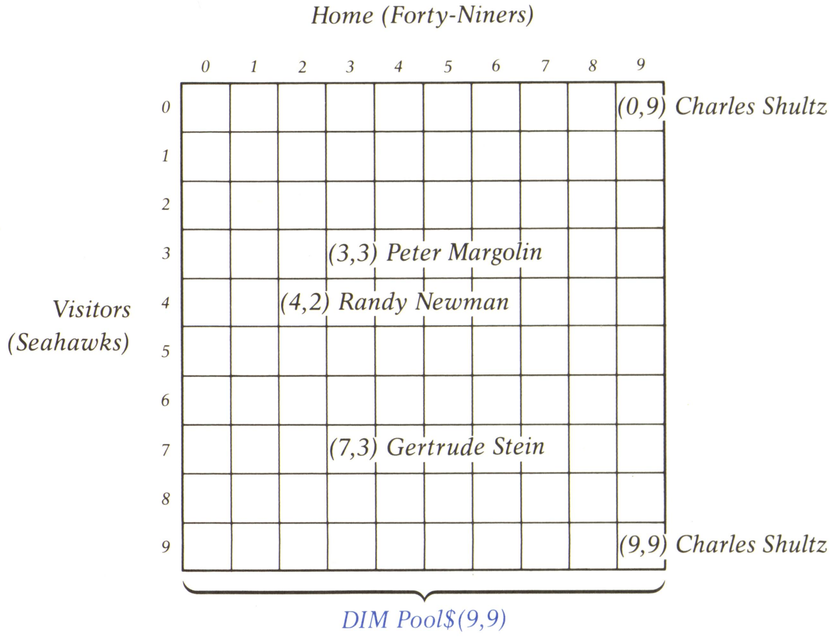
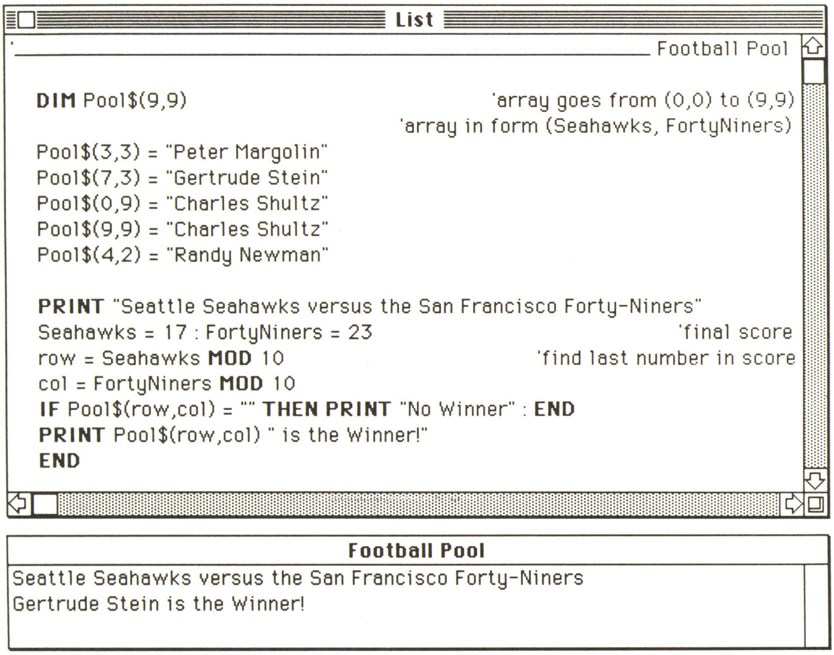

Home (Forty-Niners)
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
Visitors (Seahawks)
(0,9) Charles Shultz
(3,3) Peter Margolin
(4,2) Randy Newman
(7,3) Gertrude Stein
(9,9) Charles Shultz
DIM Pool$(9,9)

FIGURE 9-8

A football-pool card as an example of a two-dimensional array



```
'_________________________________________________ Football Pool

DIM Pool$(9,9)                          'array goes from (0,0) to (9,9)
                                   'array in form (Seahawks, FortyNiners)
Pool$(3,3) = "Peter Margolin"
Pool$(7,3) = "Gertrude Stein"
Pool$(0,9) = "Charles Shultz"
Pool$(9,9) = "Charles Shultz"
Pool$(4,2) = "Randy Newman"

PRINT "Seattle Seahawks versus the San Francisco Forty-Niners"
Seahawks = 17 : FortyNiners = 23                          'final score
row = Seahawks MOD 10                      'find last number in score
col = FortyNiners MOD 10
IF Pool$(row,col) = "" THEN PRINT "No Winner" : END
PRINT Pool$(row,col) " is the Winner!"
END
```

```
Seattle Seahawks versus the San Francisco Forty-Niners
Gertrude Stein is the Winner!
```

FIGURE 9-9

A program to compute the winner of a football pool

Each person bets on one or more of the ending-score combinations located in the cells of the card. Only one person can bet on each combination, so there is only one winner on a football-pool card. If all the cells on the card are sold for one dollar each, the winner will get $100 (minus an agent's fee, of course).

Figure 9-9 shows a program that simulates the football-pool card shown in Figure 9-8. In this program the Seattle Seahawks are playing the San Francisco Forty-Niners, and the Seahawks are the visitors.

The program nicely illustrates how easy it is to represent an external organization of two-dimensional data in a BASIC program using arrays. The football card and its cells should remind you of the *BigFoot(1,5)* two-dimensional array we studied earlier, which was a 2-by-6 array. The football card is a 10-by-10 array. Can we use a BASIC array to simulate the card? Sure. What needs to be stored on the card? The name of the person betting on a particular cell. Names are best represented with strings in BASIC, so naturally we need a string array.

The program defines a string array called *Pool$(9,9)*. This is done in the DIM statement. Next we need to fill the array with the names of the bettors. The assignment statements following the DIM do this. Look how easy it is to place a bet. Charles Shultz bets on the number pair (0,9) with this statement:

```
Pool$(0,9) = "Charles Shultz"
```

This simply fills element (0,9)—row 0, column 9—of the array *Pool$* with the name *Charles Shultz.* Once the names have all been recorded in the array, the program presents the final score, and then uses the MOD function to identify the scores' last digits (MOD simply returns the remainder after a division, so in the example, if *Seahawks* is 17, then 17 ÷ 10 is 1 with a remainder of 7, and 7 is assigned to the variable *row*). The variable *row* now contains the winning row (visitors' last digit) and the variable *col* contains the winning column (home team's last digit). The IF...THEN statement at the end of the program simply takes *row* and *col* and uses them to examine the contents of *Pool$(row,col).* If there is no string there, then no one bet that particular cell and there is no winner. If there is a string there, then the string name found in the array is displayed with the message: *is the Winner!* Try adding more names to the football pool and running the program a few times with different scores to see how it works.

THE OPTION BASE STATEMENT

Earlier we said you might want to number your array subscripts starting from 1, and ignore the 0th element, since it often makes the program more obscure. The designers of Microsoft BASIC must have felt this way too, because they provided a statement that controls what the lowest number in an array subscript may be: 0 or 1. The statement is called OPTION BASE and it works like this:

OPTION BASE 0 makes the lowest element of all arrays 0
OPTION BASE 1 makes the lowest element of all arrays 1

When we first start our program, if it doesn't use OPTION BASE, the lowest subscript we can use is 0. If we include OPTION BASE 1, it changes the lowest possible subscript to 1 (any number other than 0 or 1 following OPTION BASE will cause a syntax error). A subscripted variable that tries to access the 0th element after an OPTION BASE 1 will produce an error and stop the program.

We can write a program that ignores the 0th element of the array, as we have, and never use OPTION BASE 1. But by including it in our program, we let any other programmer know that 1 is the lowest usable subscript element.

THE ERASE STATEMENT

Suppose we wanted to change the size of an array after it has already been defined by a DIM statement. Can we use another DIM statement, like this?

```
DIM Cheerios(100,100)
:
[intervening program lines]
:
DIM Cheerios(5,2)
```

The answer is a resounding, "No, but..." We would get an error message if we tried this sequence *unless* we first erased the array. The keyword ERASE in Microsoft BASIC performs this task. If we use the statement *ERASE array name,* the name and contents of the previously dimensioned array will be erased. After the ERASE statement, the erased array no longer exists, the space it occupied is now available to our program, and the array name can be used afresh in another array. So, if we modified our previous example like this:

```
DIM Cheerios(100,100)
:
[intervening program lines]
:
ERASE Cheerios
DIM Cheerios(5,2)
```

the size of the *Cheerios* array would become 6 by 3 (don't forget that element *0* is a part of the array).

Now that we are experts on arrays, it's time to learn about a few closely associated statements that make arrays even more powerful in our programming. The new statements will make it much easier to load an array with information than it was with the assignment statements we have used in this chapter.

THE READ, DATA, AND RESTORE STATEMENTS

When this chapter began, we used arrays to organize the information processed by our programs. Our chapter on subroutines stressed the concept of *structuring* programs to make them easy to understand and easy to modify. Are you beginning to sense a recurring theme here? *Organization, structure, format:* These are the ideas that are repeated over and over in the programming environment. But at this point a flaw still exists in this structuring. The flaw has to do with how we get information into our program for processing.

There are two basic ways to get information into a program at this point: *interactive entry* (such as the INPUT or MOUSE statements), or *static assignment statements* within the program itself. For example, both of the following statements assign the title for a graph to the same variable:

```
INPUT "Enter the Title "; Title$
Title$ = "Number of square pegs"
```

Interactive entry forces us to type in new information each time the program is run. Entry of information by assignment makes the information available to the program automatically each time it's run.

We used the assignment method in our examples involving the Brothers Four Corporation. The names of the employees, their rates of pay, and hours worked were entered into respective variables in the program by assignments like:

```
Hours(1) = 44
Rate(1) = 12
EmpName$(1) = "Peter Margolin"
```

This worked okay for our short program, but imagine having to enter array information like this for a company with 300 employees—it would take 300 assignment statements! But don't despair: We can solve this problem with the DATA and READ statements.

The DATA and READ statements allow us to assign information to variables in our programs in a simple and efficient manner. They save us lots of typing, and they work with both strings and numbers. Let's see how.

The DATA statement

A DATA statement is a program line preceded by the keyword DATA. Following the word DATA is a list of string or numeric constants separated by commas, like this:

```
DATA "KGO", "Beat me to the punch", 32.555, 3.14159
DATA 100, 103, 105, 107, 109, 109, 110
DATA "Go, Ye of Weakness", 1111, "Seattle"
```

The word DATA is non-executable, meaning our BASIC program will not do anything when the statement is first encountered in the program (although the information in the DATA statement will be registered). There can be several DATA statement lists in a program. How do we use this information? The READ statement is used to access the information stored in a DATA statement.

The READ statement

READ accesses lists of information contained in the DATA statements, and assigns this information to appropriate variables: string constants to string variables, numeric constants to numeric variables. One or more variables follow the READ keyword, for example:

```
READ radio
READ sayWhat$, dozenRabbits
```

Like the DATA statement, commas are used with READ to separate items, but in this case the items are variables instead of constants. Let's see how READ and DATA work together:

```
READ radio, pi, sayWhat$
:
[intervening program lines]
:
DATA 810, 3.14159, "Beat me to the punch"
```

When the statement *READ radio* is executed, BASIC goes to the first item in the DATA statement list, here the number *810,* and assigns that value to the variable *radio.* This is the same as typing *radio = 810.* When the next READ variable is encountered, the *next item* in the DATA list is assigned to that variable. So in the example, the constant *3.14159* is assigned to *pi,* which is the next variable in the READ statement.

This process repeats until all the variables in the READ statement have been filled or assigned a data item.

The use of DATA is not restricted to numeric variables. We can also assign strings to string variables. In our previous example the last variable in READ is the variable *sayWhat$*. The corresponding item in the DATA statement is *Beat me to the punch.*

If we're setting each element in a large array to a string, it can get tedious to type the quotation marks around every string. Because of this, the handling of DATA statements in BASIC has been set up so that the quotes are optional. The commas are actually all we need to separate string constants. Therefore, the two statements:

```
DATA "California", "Wisconsin", "New York"
DATA California, Wisconsin, New York
```

are equivalent in Microsoft BASIC. But what if we wanted to have a comma appear *inside* a string to be read and subsequently printed on the screen? Consider these two DATA statements:

```
DATA Smith, Mark, Hari, Mata
DATA "Smith, Mark", "Hari, Mata"
```

The first DATA statement represents four separate strings, while the second represents two. In general, we're safer enclosing strings in DATA statements with quotes.

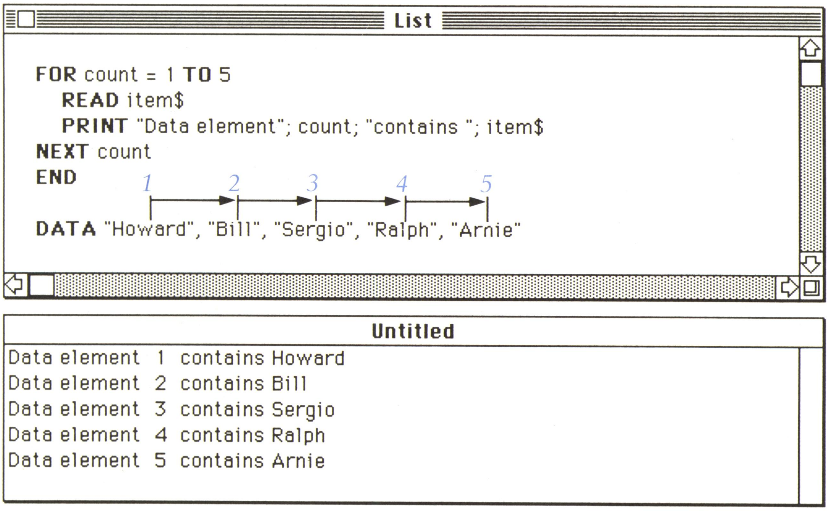
The data pointer

You may be wondering how the READ statement keeps track of which DATA item will be assigned to which variable. It is actually quite simple: BASIC assumes that the list is continuous. Each time a READ statement is executed, a "data pointer" inside BASIC is incremented to point to the next DATA item.

In the example in Figure 9-10, the program executes the *READ item$* statement five times in a loop. The first time through the loop, when *count = 1,* the data pointer is pointing at the first element, here the string *Howard.* So when *READ item$* is encountered, the string *Howard* is assigned to *item$* and printed out. The next time READ is executed, the data pointer is "bumped" and the string *Bill* is assigned to *item$.*

FIGURE 9-10

The data pointer is incremented each time a data element is assigned to a variable



```
FOR count = 1 TO 5
   READ item$
   PRINT "Data element"; count; "contains "; item$
NEXT count
END

DATA "Howard", "Bill", "Sergio", "Ralph", "Arnie"
```

```
Data element  1  contains Howard
Data element  2  contains Bill
Data element  3  contains Sergio
Data element  4  contains Ralph
Data element  5  contains Arnie
```

What if there is more than one variable in the READ statement? Look at this example:

```
READ a, b
READ c
DATA 111, 222, 333, 444, 555
```

In this situation, BASIC assigns *111* to *a*, *222* to *b*, and *333* to *c*. Nothing will happen with *444* and *555*. But the BASIC data pointer is now at the beginning of *444*, so the next time a READ statement occurs in the program it will pick up *444*. When there is more than one variable in a READ statement, or when there is more than one READ statement, BASIC assigns the first item to the first READ variable, then moves to the second READ variable and assigns it the next item, and so on.

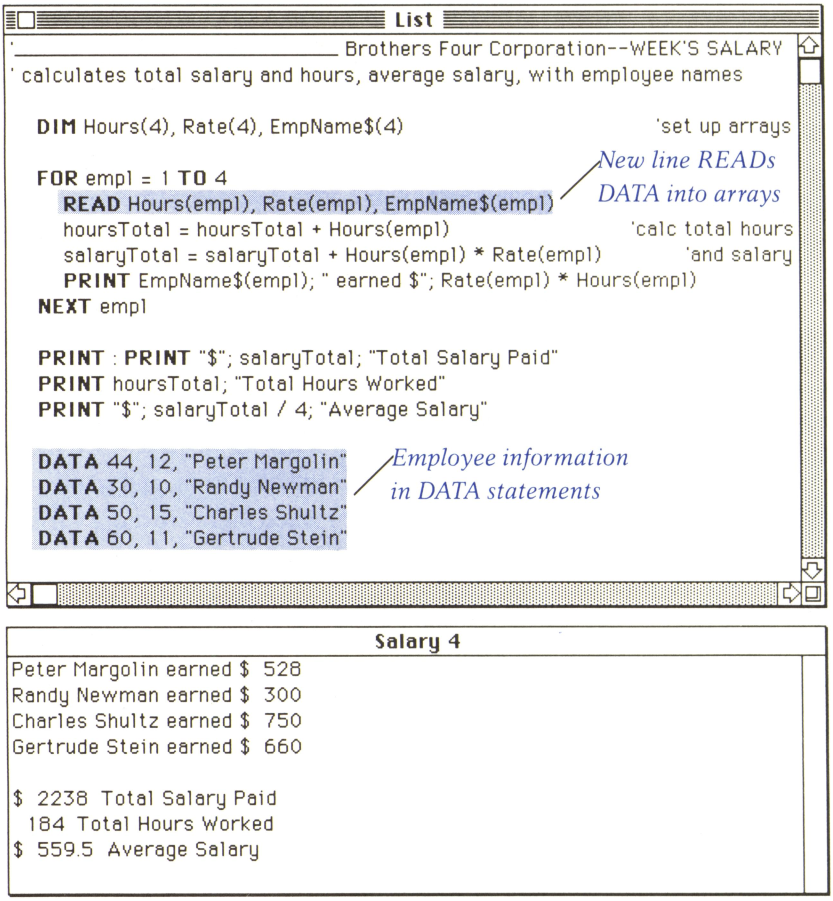
What happens when there are more variables than there are items in the DATA statement? For example:

```
READ diamonds, clubs, hearts, spades
DATA 12, 43
```

A situation like this produces what is called an *OUT OF DATA* error. An error box appears on the screen and the program stops. This is a programming error caused by not carefully matching READ variables with DATA constants. It is up to the programmer to ensure this doesn't happen. This type of error can frequently occur because, as we shall soon see, it is easy to miscount the number of items in the DATA list and not have enough.

Let's take our Brothers Four Corporation program, the last one that included strings, and see if we can smooth it out by using DATA statements. Recall that we stored our employees' information in three simple arrays, *Hours(4)*, *Rate(4)*, and *EmpName$(4)*. Figure 9-11 shows how we modified the program to use READ and DATA.

Notice how nicely the information is formatted at the end of the program. Not only is the information more compactly stored, it is much easier for the programmer to enter it. Examine the FOR...NEXT loop that does the actual assignments. All we had to do was add the READ line to get the program to read the DATA statements and assign the values to the array variables. This example shows most clearly the value of the DATA



```
'_______________________________ Brothers Four Corporation--WEEK'S SALARY
' calculates total salary and hours, average salary, with employee names

  DIM Hours(4), Rate(4), EmpName$(4)                          'set up arrays

  FOR empl = 1 TO 4
    READ Hours(empl), Rate(empl), EmpName$(empl)
    hoursTotal = hoursTotal + Hours(empl)                   'calc total hours
    salaryTotal = salaryTotal + Hours(empl) * Rate(empl)        'and salary
    PRINT EmpName$(empl); " earned $"; Rate(empl) * Hours(empl)
  NEXT empl

  PRINT : PRINT "$"; salaryTotal; "Total Salary Paid"
  PRINT hoursTotal; "Total Hours Worked"
  PRINT "$"; salaryTotal / 4; "Average Salary"

  DATA 44, 12, "Peter Margolin"
  DATA 30, 10, "Randy Newman"
  DATA 50, 15, "Charles Shultz"
  DATA 60, 11, "Gertrude Stein"
```

New line READs DATA into arrays

Employee information in DATA statements

```
Peter Margolin earned $  528
Randy Newman earned $  300
Charles Shultz earned $  750
Gertrude Stein earned $  660

$  2238  Total Salary Paid
  184  Total Hours Worked
$  559.5  Average Salary
```

FIGURE 9-11

Employee hours, pay rates, and names incorporated in DATA statements statement. Suppose we added 20 employees to the company. All we would have to do is change the numbers in the DIM statement, the terminal value of the loop counter, and add the new DATA items. If we needed more room, we could store two or three employees on one line:

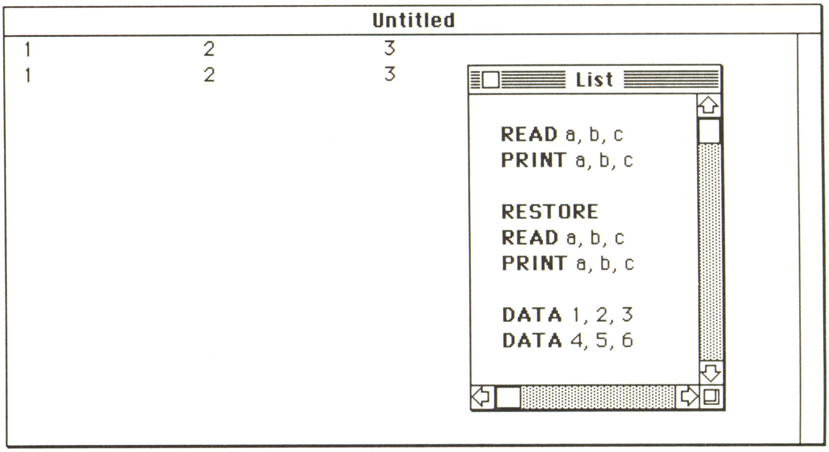
```
DATA 44, 12, "Peter Margolin", 30, 10, "Randy Newman"
```

DATA statements are frequently used in BASIC programs. You see them used to hold constant data that the programmer wants easy access to. For example, DATA statements may contain the coordinates of graphics, the frequency and volume for music using the SOUND statement, constants for structural analysis, or even numbers representing a 68000 microprocessor program.

The RESTORE statement

Sometimes it is useful to READ the information in a DATA statement a second or third time. This may be the case when you don't wish to use a single large array to hold all the values. Since all of the values in the data statements are fixed, adding one large array to a program consumes an equally large amount of memory. If you can simply reuse the information in the DATA statements the need for an array might be unwarranted. Typical situations might be a long line of musical notes stored as DATA statements that you wish to play several times, a collection of default values for a word-processing program, or a number of numeric constants you need to use in different calculations.

In order to reread a list of items from a DATA statement that have already been read, we need a way to reset the data pointer to the beginning of the *first* item in the first DATA statement. If we don't do this, the pointer will be incremented from its last position, probably giving an error. The RESTORE statement is used to solve this problem. The program in Figure 9-12, on the following page, shows how RESTORE is used. The first READ statement reads values from the DATA statements and prints them out. The pointer now points at number 4 in the second DATA statement. RESTORE then resets the pointer to the first item in the first DATA statement, here 1, and thus the second READ reads the same data again.



```
1                    2                    3
1                    2                    3
```

```
  READ a, b, c
  PRINT a, b, c

  RESTORE
  READ a, b, c
  PRINT a, b, c

  DATA 1, 2, 3
  DATA 4, 5, 6
```

FIGURE 9-12

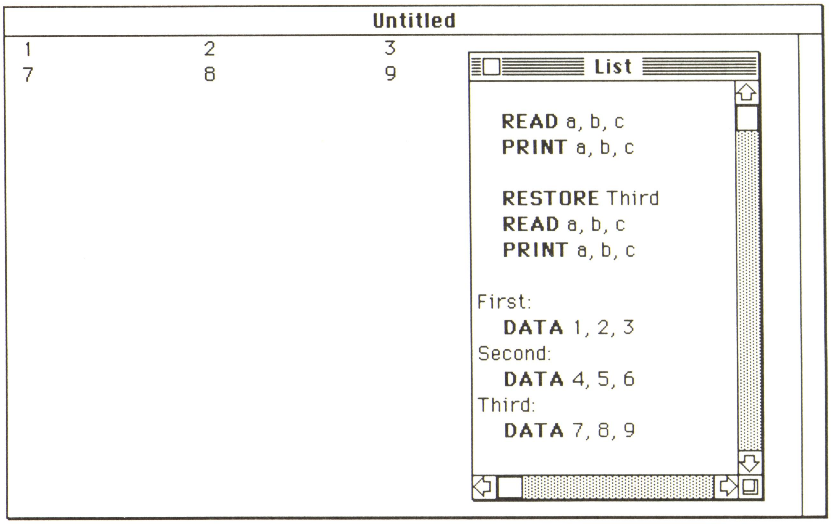
A sample program using RESTORE

RESTORE can also be told to reset the data pointer to a specific line number or label. This is useful when you have arranged the data in DATA statements such that it is grouped in sections starting at different lines. The statement:

RESTORE *label* (or *line number*)

resets the data pointer to the beginning of the DATA statement preceded by the label or line number specified in the statement.

The example in Figure 9-13 shows how this optional version of RESTORE works. Here, it causes BASIC to skip over the DATA statement with the *Second* label and read the DATA preceded by the *Third* label.



```
1                    2                    3
7                    8                    9
```

```
  READ a, b, c
  PRINT a, b, c

  RESTORE Third
  READ a, b, c
  PRINT a, b, c

First:
  DATA 1, 2, 3
Second:
  DATA 4, 5, 6
Third:
  DATA 7, 8, 9
```

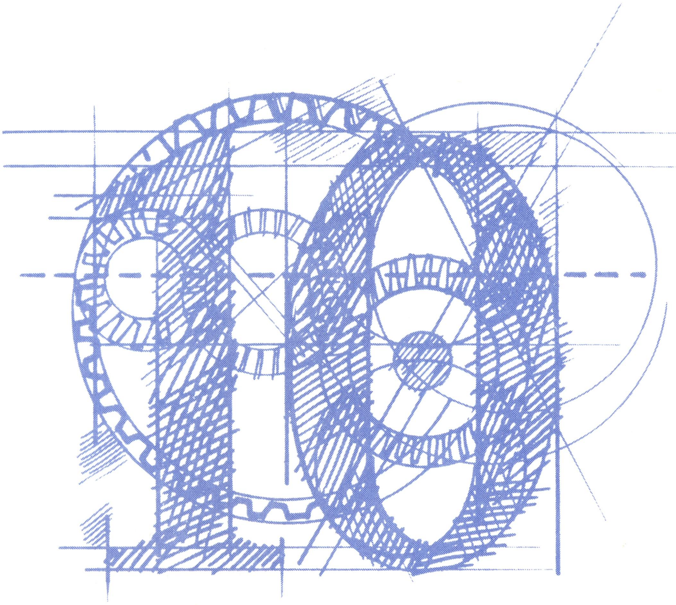
FIGURE 9-13

RESTORE used to reset the data pointer to a labeled READ statement

REFRESH-YOURSELF WINDOW

We have gone pretty deep into ways to structure infomation and data in a BASIC program. We have learned about one of the most powerful of BASIC features: arrays. We know how to use a single and multidimensional array, how the array is organized, how to DIMension an array, and what OPTION BASE is. Subscripts and array mathematics were covered. Then we saw how to ERASE an array so it could be redimensioned later. Finally, we found out how to nicely fill arrays (or any variables for that matter) with constant data items using READ and DATA.

In the next chapter, we will turn to another important element of BASIC: *strings.*



Strings

We have already used strings on numerous occasions in this book. We have learned that the BASIC statement PRINT "Good Morning" *causes the phrase* Good Morning *to appear on the screen, and that this phrase is called a* string, *or more precisely, a* string constant. *We've also learned that variables can be used to hold string values. Thus the BASIC statement* lastName$

= *"Smith"* assigns the value *Smith* to the variable *lastName$,* and *PRINT lastName$* causes *Smith* to be printed on the screen.

However, the ways that strings have been used so far are only the tip of the iceberg. There are many functions designed specifically for use with strings, and a wide variety of problems can be solved using strings together with these functions. In this chapter we'll explore some of these string-specific functions. We'll show you how to divide strings into parts and put them back together, and we will show you the "code" that BASIC uses to store and operate with strings. We'll learn how to convert characters to numbers and back again, how to search for a word in a sentence, and all sorts of other useful and powerful functions to extend the power of our programs.

HOW BASIC THINKS OF STRINGS

A string is called a string because it consists of a number of separate characters which are "strung" together. A character, as we've learned, can be a letter (upper- or lowercase), a digit (from 0 to 9), or any of the punctuation marks you can find on your keyboard. Later in this chapter we'll see that there are other characters recognized by Microsoft BASIC that are used for special purposes like sounding a beep or causing the current print location to "tab" to a fixed position. But primarily we can think of characters as being letters, numbers, and punctuation marks.

It's helpful to think of a string as occupying a series of memory locations in our computer. Each location holds one character of the string. You can think of the memory locations as little "bins," and the characters that make up the string as being objects that occupy the bins, as shown in Figure 10-1. The memory locations, or bins, are fixed; but the letters can be moved around between the bins any way we want. One of our goals in this chapter is to show you how to move the characters between the bins.

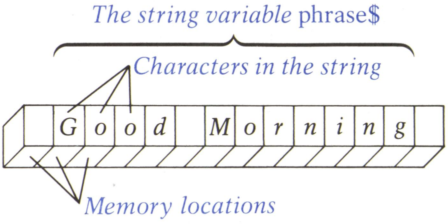


FIGURE 10-1

A string occupying a series of memory locations

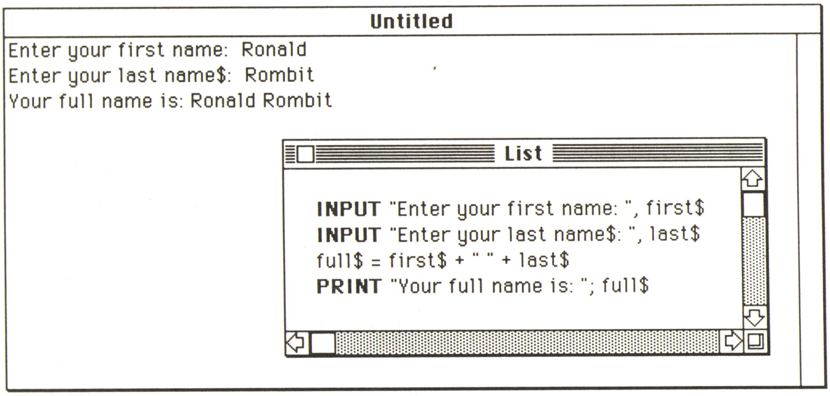
BASIC keeps track of where strings are in memory, and how many characters are stored in each string. Suppose we assigned the string shown in Figure 10-1, *Good Morning,* to the variable *phrase$*. Thereafter, when we referred to this variable in our program, BASIC would know where to look to find the appropriate string of characters.

PUTTING STRINGS TOGETHER: THE + SIGN

One of the simplest things we can do with strings is to put two or more of them together to form another, longer, string and Figure 10-2 shows a program that does exactly that.

The first two lines of the program prompt the user to enter his or her first and last name, and assigns them to the variables *first$* and *last$*. The third line of the program is where the action is. Using the + sign, we form a new string, called *full$,* which consists of *first$,* then a space, and finally *last$*. (The space is necessary to keep the two names from crowding up against each other, as in *RonaldRombit.*)

Note that the string variables *first$* and *last$* are not changed, so their characters still occupy their little bins in memory. However, a third string has been created: *full$*. The string constant " " (the space) doesn't occupy a bin in memory, since it's a constant and not a variable; only variables are explicitly given space in memory. BASIC simply takes the constant (in this particular case the space) from the program, uses it, and then forgets about it.



```
Enter your first name:  Ronald
Enter your last name$:  Rombit
Your full name is: Ronald Rombit
```

```
INPUT "Enter your first name: ", first$
INPUT "Enter your last name$: ", last$
full$ = first$ + " " + last$
PRINT "Your full name is: "; full$
```

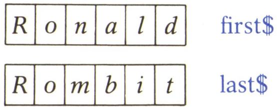
FIGURE 10-2

Using the + sign to combine strings

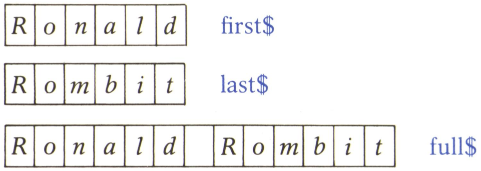
FIGURE 10-3

Concatenating strings

a) Before



b) After executing full$ = first$ + " " + last$



Putting strings together in this way is called *concatenation,* which means "chaining together"—not a bad description of the process, which is illustrated in Figure 10-3.

TAKING STRINGS APART

If we can put strings together, can we take them apart? Yes, but separating them is somewhat more complicated than putting them together because there are a lot more ways to divide a string into smaller strings than there are to put strings together. We'll be looking at three BASIC functions that take strings apart.

The LEFT$ and RIGHT$ functions

The simplest functions for getting parts of strings are LEFT$ and RIGHT$. As their names imply, LEFT$ lets you get some of the characters from the left side, or the beginning of a string, while RIGHT$ lets you get some of the characters from the right side, or the end of a string. These functions need to know two things: which string you want to take the characters from, and how many characters to get.

For example, suppose we have a string variable *phrase$* with the value *wash n' dry.* The function *LEFT$(phrase$,4)* will return the string *wash,* since these are the four leftmost characters of the string *phrase$.* Similarly, *RIGHT$(phrase$,3)* will return the string *dry,* since these are the three rightmost characters of the string *phrase$.* These operations are shown in Figure 10-4.

Of course, it doesn't matter whether a string of characters is composed of letters or the digits from 0 to 9. Figure 10-5 shows a program that

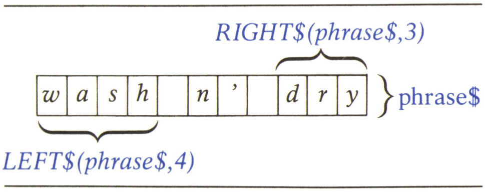


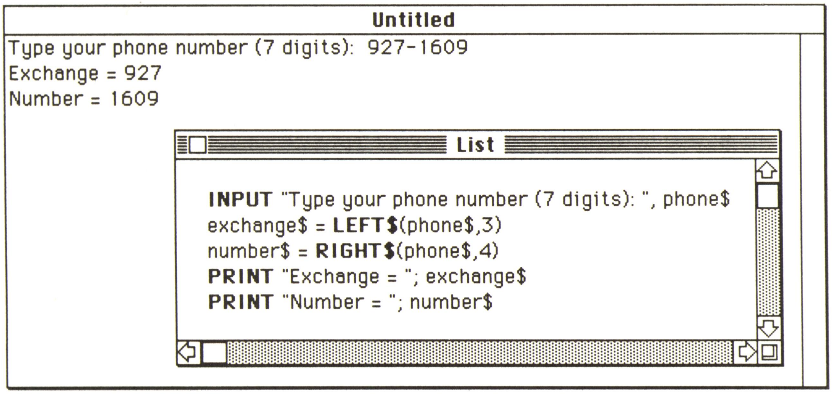
FIGURE 10-4

The effect of LEFT$ and RIGHT$

uses LEFT$ and RIGHT$ to get characters from a string composed mostly of digits: namely, a phone number.

In this program we prompt the user to enter a number, then assign it to the variable *phone$*. Then we use LEFT$ and RIGHT$ to copy from it the exchange (the first three digits) and the number (the last four digits).

If working with phone numbers seems too mundane, consider the program in Figure 10-6, which uses LEFT$ and RIGHT$ to turn discord

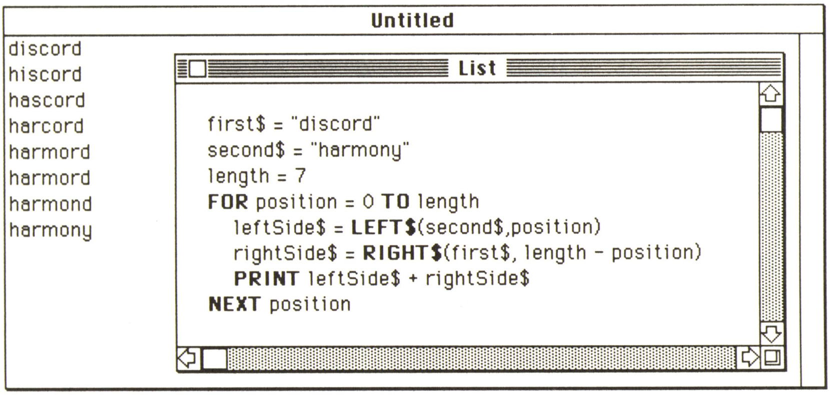


```
Type your phone number (7 digits):  927-1609
Exchange = 927
Number = 1609
```

```
INPUT "Type your phone number (7 digits): ", phone$
exchange$ = LEFT$(phone$,3)
number$ = RIGHT$(phone$,4)
PRINT "Exchange = "; exchange$
PRINT "Number = "; number$
```

FIGURE 10-5

Getting characters from a seven-digit phone number



```
discord
hiscord
hascord
harcord
harmord
harmord
harmond
harmony
```

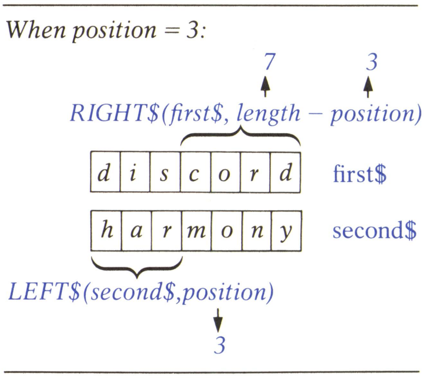
```
first$ = "discord"
second$ = "harmony"
length = 7
FOR position = 0 TO length
   leftSide$ = LEFT$(second$,position)
   rightSide$ = RIGHT$(first$, length - position)
   PRINT leftSide$ + rightSide$
NEXT position
```

FIGURE 10-6

Turning discord into harmony

FIGURE 10-7

The operation of the harmony-discord program



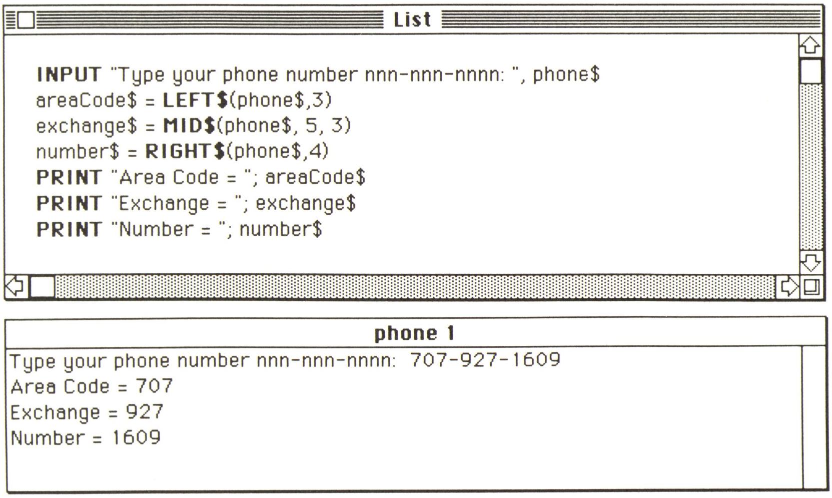
into harmony. This program demonstrates that you can use numeric variables, rather than specific constants such as 3 and 4, to tell how many characters to return in the functions LEFT$ and RIGHT$.

We start the harmony-discord program with two strings, *discord* and *harmony,* and a *pointer*—the variable *position*—pointing at the beginning of both words. Each time through the loop we copy some of the left side of the second word, *harmony,* and some of the right side of the first word, *discord.* At the beginning, *position* is 0, so we don't copy any characters from *harmony*; they're all from *discord.* The next time through the FOR loop, we get the *h* from *harmony* and the *iscord* from *discord.* And so on, until one word has been entirely transformed into the other. Note that we use the + sign to concatenate the parts of the two words before printing out the compound word.

Figure 10-7 shows how this process looks when the variable *position* has the value 3 and the word being printed is *harcord.*

The MID$ function

Now we know how to get the left or right parts of a string, but how about the middle? As you may have guessed, there is a BASIC function that lets us copy a string from the middle of another string. This function is MID$. Although slightly more complicated than LEFT$ and RIGHT$, it's also more versatile. MID$ needs to know three things: which string to use, where to start copying the characters, and how many to get. Thus, if *phrase$ = "wash n' dry",* then the function *MID$(phrase$, 6, 1)* will return the string *n,* which is a string only one character long, starting at position 6, in the string *phrase$.*



```
INPUT "Type your phone number nnn-nnn-nnnn: ", phone$
areaCode$ = LEFT$(phone$,3)
exchange$ = MID$(phone$, 5, 3)
number$ = RIGHT$(phone$,4)
PRINT "Area Code = "; areaCode$
PRINT "Exchange = "; exchange$
PRINT "Number = "; number$
```

```
Type your phone number nnn-nnn-nnnn:  707-927-1609
Area Code = 707
Exchange = 927
Number = 1609
```

FIGURE 10-8

Taking apart a ten-digit phone number

Figure 10-8 shows a program that uses MID$ to help break a ten-digit phone number into three separate parts. This program is much like the earlier one that disassembled a seven-digit number, except that the function *MID$(phone$, 5, 3)* returns the three digits from the middle of the phone number, starting at position 5, that constitute the exchange part of the number. As before, RIGHT$ is used to get the number itself, while LEFT$ is used to get the area code.

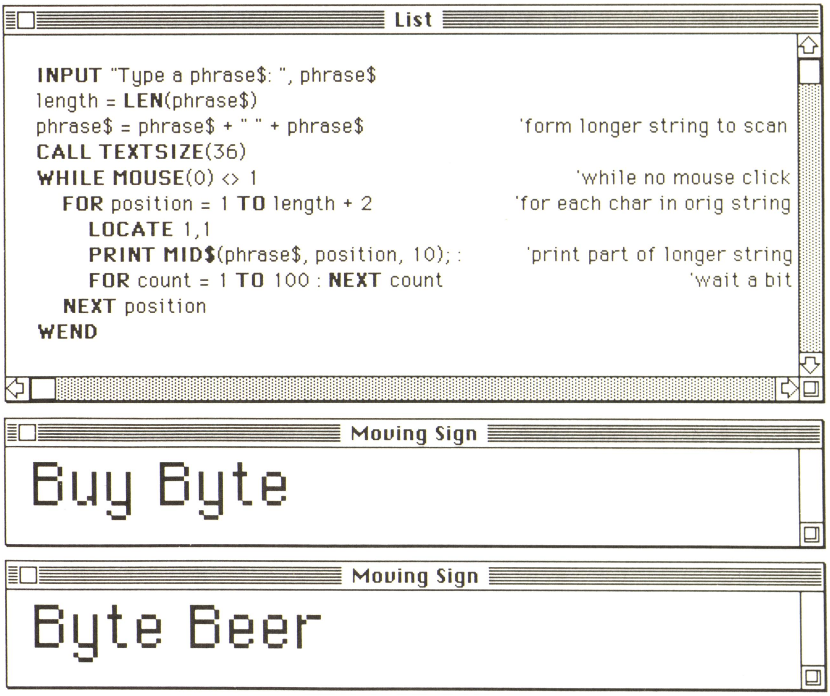
The LEN function

As we'll see in the following program, it's often useful to be able to figure out how many characters a string contains. Fortunately, there's a very nice function that does exactly this: LEN. Suppose we have a string variable *phrase$*, which has the value *wash n' dry*. There are 11 characters in this string, counting the spaces (which must be counted, since they are characters too), so the function *LEN(phrase$)* will return the value 11.

You've no doubt seen signs, such as those in Times Square in New York City, which are composed of light bulbs that turn on and off in such a way that they create the effect of moving messages. In the program in Figure 10-9 on the next page, we use the function LEN, as well as MID$ and concatenation, to create a "moving sign" effect out of any phrase we type in. Type in this program and try it out. When it starts, you can type in any phrase you like. The one in the example is *Buy Byte Beer.*

FIGURE 10-9

The Moving-Sign program



```
INPUT "Type a phrase$: ", phrase$
length = LEN(phrase$)
phrase$ = phrase$ + " " + phrase$              'form longer string to scan
CALL TEXTSIZE(36)
WHILE MOUSE(0) <> 1                                'while no mouse click
  FOR position = 1 TO length + 2               'for each char in orig string
    LOCATE 1,1
    PRINT MID$(phrase$, position, 10); :      'print part of longer string
    FOR count = 1 TO 100 : NEXT count                       'wait a bit
  NEXT position
WEND
```

```
Buy Byte
```

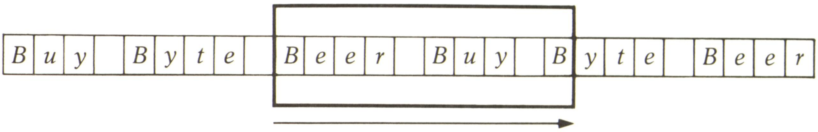
```
Byte Beer
```

The program first uses the LEN function to find out the length of the string that was typed in response to the INPUT statement; it will need this information later. The program then takes this string and concatenates it with itself to form a string twice as long. Then, after setting the text size to yield 36-point letters, the program enters a pair of nested loops. The outer WHILE...WEND loop simply ensures that the display will keep going indefinitely (until you stop it by clicking the mouse button once). It's the inner FOR...NEXT loop that does the work.

What this program does is use MID$ to take a ten-character string from the middle of the newly formed double string, display it, then take another ten-character string (but starting this time one character to the right of the last one), display it, and so forth. If you imagine a sort of frame, ten characters long, moving across the string, you can visualize how this works. Figure 10-10 shows what such a frame might look like.

FIGURE 10-10

The operation of the Moving-Sign program



The line *FOR count = 1 TO 100 : NEXT count,* the program's delay loop, slows the program so the sign can be read in each position.

THE ASCII-CODE SYSTEM

Earlier we said that you could think of strings as characters—letters, punctuation, spaces, and so forth—occupying sequences of memory locations. This may be a useful way to visualize strings, but it isn't completely accurate. The fact is, the computer doesn't actually store letters or punctuation marks themselves. The only thing the computer can store in one memory location is a *number,* which must be between 0 and 255. So in order to store the letter *A,* for example, it must store the number that *represents* the letter. It turns out that the number that represents the capital letter *A* is 65. All the other letters, both upper- and lowercase, all the punctuation marks, and even the digits from 0 to 9, are represented by different numbers. These numbers are called ASCII codes (ASCII stands for *American Standard Code for Information Interchange*). The ASCII system is the standard way that computers store and communicate characters.

Now it just so happens that all of the most-used characters—the 26 uppercase letters, 26 lowercase letters, the 10 digits, the punctuation marks, and so on—have been assigned a specific ASCII code between 0 and 127. We can rely on an *A* having code 65, no matter what program we are using on what computer. Things get a little more complicated, though, for the ASCII numbers from 128 through 217 because no characters have been explicitly assigned to them. The result is a kind of ASCII free-for-all, with different companies assigning completely different characters to this range of codes. For example, Apple has assigned a variety of special characters to these ASCII numbers, such as Greek letters and mathematical symbols. Although it's not strictly accurate to refer to these numbers as ASCII codes (because they are not industry-wide standards), the principle is the same: The numbers are used to represent different characters. However, for convenience we'll refer to all of these numeric codes, whether they're really ASCII or an invention, as ASCII codes.

Why do you need to know how the computer stores letters? It's true that you can do many things with strings without knowing these codes. In some cases, though, it's useful to understand what's actually going on inside the computer, as we'll see.

The CHR$ function

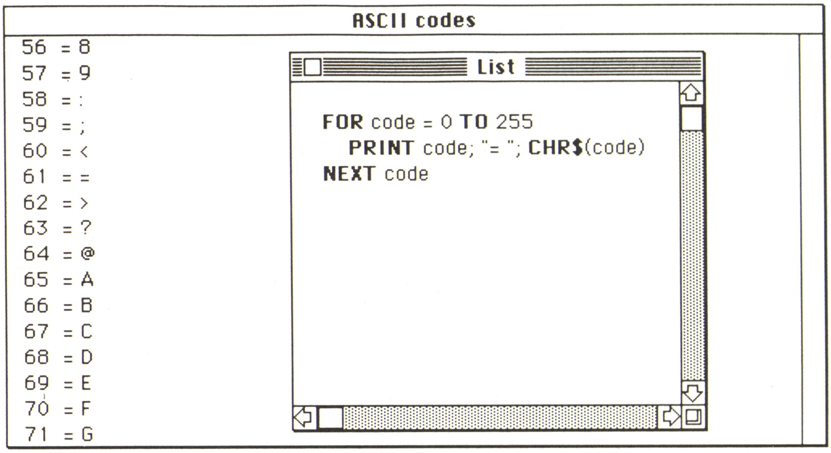
How can we find out what character is represented by a particular ASCII code? The CHR$ function will take any number between 0 and 255 and return the corresponding character. Figure 10-11 shows a simple program that causes all the ASCII codes and their corresponding characters to be printed in the Output window.

As you can see, this program is a simple FOR...NEXT loop that runs from 0 to 255. Each time through the loop, the program prints the next number in the counter variable *code,* and also the character that number represents. The character is returned by the *CHR$(code)* function. (If you want to *suspend* the program to examine a particular section of the display, press the Command and S keys at the same time, then the Spacebar to continue; or just choose Suspend and Continue from the Run menu.)

When you run the program you'll notice that many of the codes with numbers from 0 to 32 do strange things, such as print empty boxes. That's because this range of codes is used for non-printable "characters," such as the beep, carriage return, linefeed, formfeed, tab, and so on. Also, numbers above 217 in most fonts have not been assigned characters, and empty boxes are printed for these numbers as well.

You can also generate a table of ASCII codes that you can print out and pin up on your wall for quick and easy reference. The program shown in Figure 10-12 does just that.

This program is similar to the previous one, except that it prints more than one code on each line, and uses the variable *column* to keep

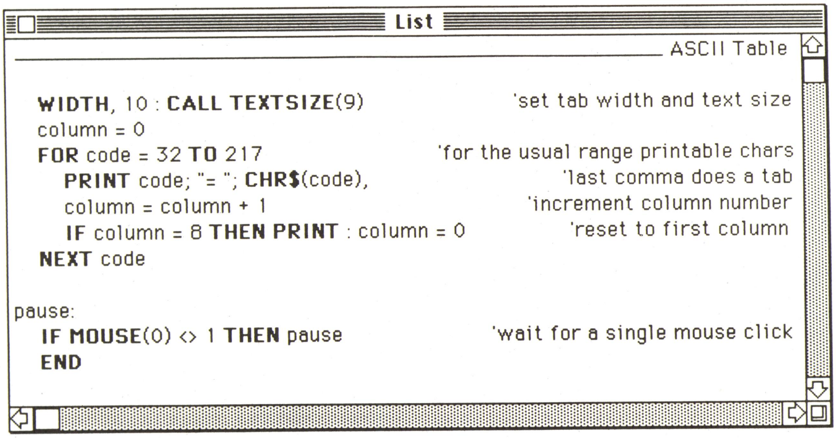


```
56  = 8
57  = 9
58  = :
59  = ;
60  = <
61  = =
62  = >
63  = ?
64  = @
65  = A
66  = B
67  = C
68  = D
69  = E
70  = F
71  = G
```

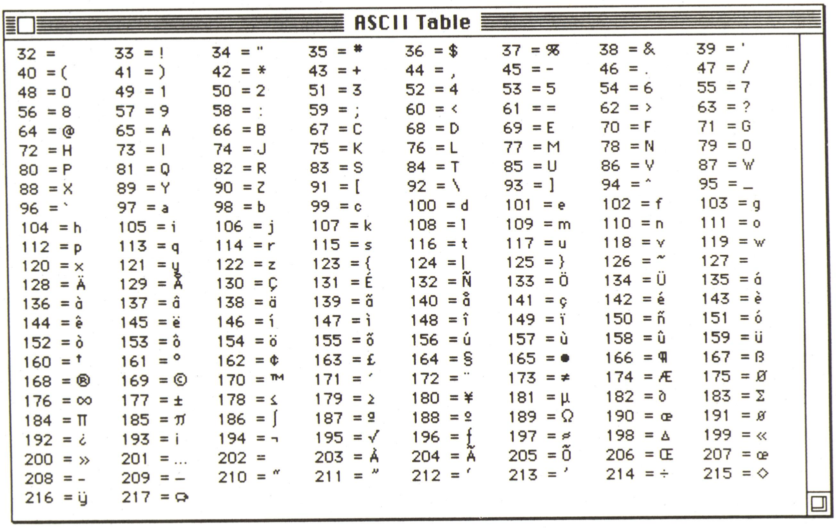
```
FOR code = 0 TO 255
  PRINT code; "= "; CHR$(code)
NEXT code
```

FIGURE 10-11

A simple ASCII-code program



```
_______________________________________________________________ ASCII Table

  WIDTH, 10 : CALL TEXTSIZE(9)                    'set tab width and text size
  column = 0
  FOR code = 32 TO 217                      'for the usual range printable chars
     PRINT code; "= "; CHR$(code),                       'last comma does a tab
     column = column + 1                           'increment column number
     IF column = 8 THEN PRINT : column = 0                'reset to first column
  NEXT code

pause:
  IF MOUSE(0) <> 1 THEN pause                   'wait for a single mouse click
  END
```



```
32  =       33  = !     34  = "     35  = #     36  = $     37  = %     38  = &     39  = '
40  = (     41  = )     42  = *     43  = +     44  = ,     45  = -     46  = .     47  = /
48  = 0     49  = 1     50  = 2     51  = 3     52  = 4     53  = 5     54  = 6     55  = 7
56  = 8     57  = 9     58  = :     59  = ;     60  = <     61  = =     62  = >     63  = ?
64  = @     65  = A     66  = B     67  = C     68  = D     69  = E     70  = F     71  = G
72  = H     73  = I     74  = J     75  = K     76  = L     77  = M     78  = N     79  = O
80  = P     81  = Q     82  = R     83  = S     84  = T     85  = U     86  = V     87  = W
88  = X     89  = Y     90  = Z     91  = [     92  = \     93  = ]     94  = ^     95  = _
96  = `     97  = a     98  = b     99  = c     100 = d     101 = e     102 = f     103 = g
104 = h     105 = i     106 = j     107 = k     108 = l     109 = m     110 = n     111 = o
112 = p     113 = q     114 = r     115 = s     116 = t     117 = u     118 = v     119 = w
120 = x     121 = y     122 = z     123 = {     124 = |     125 = }     126 = ~     127 =
128 = Ä     129 = Å     130 = Ç     131 = É     132 = Ñ     133 = Ö     134 = Ü     135 = á
136 = à     137 = â     138 = ä     139 = ã     140 = å     141 = ç     142 = é     143 = è
144 = ê     145 = ë     146 = í     147 = ì     148 = î     149 = ï     150 = ñ     151 = ó
152 = ò     153 = ô     154 = ö     155 = õ     156 = ú     157 = ù     158 = û     159 = ü
160 = †     161 = °     162 = ¢     163 = £     164 = §     165 = •     166 = ¶     167 = ß
168 = ®     169 = ©     170 = ™     171 = ´     172 = ¨     173 = ≠     174 = Æ     175 = Ø
176 = ∞     177 = ±     178 = ≤     179 = ≥     180 = ¥     181 = µ     182 = ∂     183 = ∑
184 = ∏     185 = π     186 = ∫     187 = ª     188 = º     189 = Ω     190 = æ     191 = ø
192 = ¿     193 = ¡     194 = ¬     195 = √     196 = ƒ     197 = ≈     198 = ∆     199 = «
200 = »     201 = …     202 =       203 = À     204 = Ã     205 = Õ     206 = Œ     207 = œ
208 = –     209 = —     210 = “     211 = ”     212 = ‘     213 = ’     214 = ÷     215 = ◊
216 = ÿ     217 = ⚐
```

FIGURE 10-12

The ASCII-Table program and its output

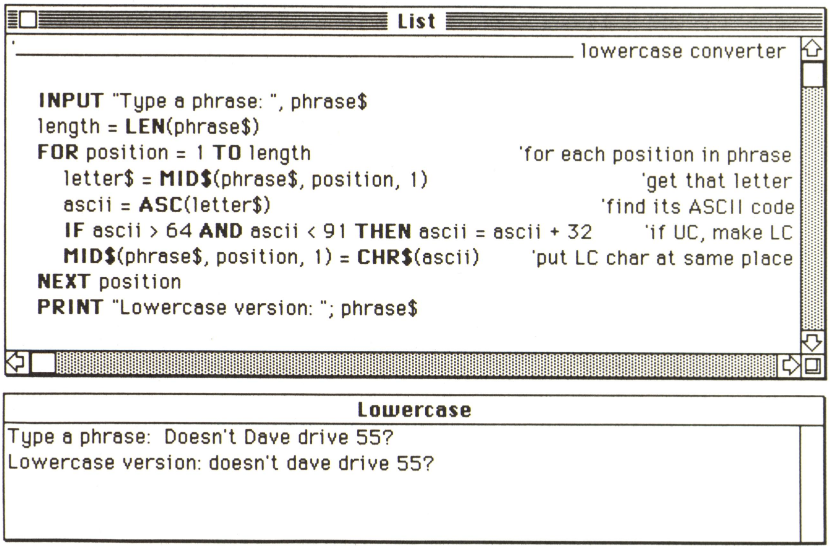
track of which column it's currently printing in. Each time *column* reaches 8, the program starts a new line with the print statement, and resets *column* to 0. The IF statement before the END pauses the program until the mouse button is clicked so that the table remains on the screen, and you can print it out using the Command-Shift-4 key combination.

Earlier in this section we mentioned that Apple created different characters for the numbers 128 to 217. You can see in the table that these include diacritical marks, foreign letters, and mathematical and other symbols. If you've used MacPaint, MacWrite, or Microsoft Word, you're probably aware that the Macintosh can print text using different *fonts.* Although we're going to discuss fonts in detail in Chapter 15, we should point out now that each font has a unique set of these special characters for numbers between 128 and 217 (with some overlapping of characters between fonts). Later on, after you've learned how to work with fonts, you might want to amend this program to generate ASCII tables for any font that you use regularly.

The ASC function

We've seen that the CHR$ function takes a number and returns the character the number represents. Can we go the other way, starting with a character and finding the ASCII code for the character? We can, with the ASC function. Let's use ASC in a program that illustrates why it can be useful to know about ASCII codes. Our program, shown in Figure 10-13 with a sample output, will take any sentence we type in, convert all the uppercase letters to lowercase, and print out the resulting string.

How does this program work? If you look at the ASCII table in Figure 10-12, you can see that the ASCII code for each lowercase letter is separated from that of the corresponding uppercase letter by exactly 32. For example, the code for *A* is 65, the code for *a* is 97, and 97 minus 65 is 32. So if we want to change an uppercase letter to lowercase, all we do is convert it to its ASCII code, add 32, and convert it back to a character.



```
' ______________________________________________ lowercase converter

INPUT "Type a phrase: ", phrase$
length = LEN(phrase$)
FOR position = 1 TO length                          'for each position in phrase
   letter$ = MID$(phrase$, position, 1)                         'get that letter
   ascii = ASC(letter$)                                      'find its ASCII code
   IF ascii > 64 AND ascii < 91 THEN ascii = ascii + 32          'if UC, make LC
   MID$(phrase$, position, 1) = CHR$(ascii)          'put LC char at same place
NEXT position
PRINT "Lowercase version: "; phrase$
```

```
Type a phrase:  Doesn't Dave drive 55?
Lowercase version: doesn't dave drive 55?
```

FIGURE 10-13

The Lowercase program

However, our program can't go through *every* character blindly and add 32 to its ASCII value. It must perform this operation only on characters that are uppercase letters. How can it tell which characters these are? By converting the characters to their ASCII values, and seeing if these values fall between 65 (*A*) and 90 (*Z*). If so, 32 is added to the ASCII code for the character. If not, the character is included in the string unchanged.

The MID$ statement

If you look at the program listing, you'll see that the keyword MID$ is used in two different ways: once as a function, and once as a statement. You're already familiar with MID$ used as a function. For example, in the line:

```
letter$ = MID$(phrase$, position, 1)
```

you know that MID$ will extract one letter from the string *phrase$* and assign it to the variable *letter$*. In this case, where it is used as a function, MID$ appears on the right side of the equal sign.

What's new is the line where MID$ is located on the *left* side of the equal sign:

```
MID$(phrase$, position, 1) = CHR$(ascii)
```

Its purpose here is not to return a value, but to actually *change an existing string*. The string to be changed, the starting position of the change, and the number of characters to be changed are specified by the parameters inside the parentheses following MID$. The string that is to replace the one specified is on the right side of the equal sign.

Our program examines each character in the string *phrase$*, one at a time, using the pointer variable *position*. The character is converted to its ASCII value in the line *ascii* = *ASC(letter$)*. If *ascii* is greater than 64 and less than 91, the character is an uppercase letter, and is changed to lowercase by adding 32 to its ASCII value. Whether it was changed or not, the character is then replaced in the string using the MID$ statement.

Notice how this program makes use of the numeric ASCII values of the characters to determine if a letter lies in a particular range. It's for this sort of operation that knowing about the ASCII codes is useful.

Converting strings: The VAL and STR$ functions

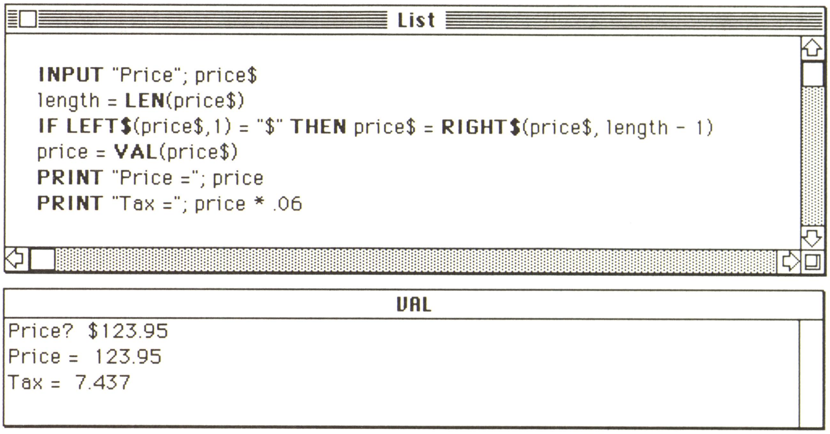
We've seen that CHR$ converts an ASCII code to its corresponding character, and that ASC does the reverse, converting a character to its ASCII code. There are two other functions, VAL and STR$, that perform similar operations, but on *strings* instead of single characters. Let's take a look at each of these.

The VAL function

The first of these functions, VAL, is useful when you have a string with a lot of numeric digits in it and you want to transform them into a single number so that you can do arithmetic with it. The simple program in Figure 10-14 shows how this works.

This program is designed to deal with the situation where our program asks for a price to be typed in, and we can't be sure that the user won't type a dollar sign before the number. Since BASIC won't accept a dollar sign as part of a numeric variable, the program assigns the price to a string variable, *price$*. It then uses LEFT$ to examine the first character of the string. If this character is a dollar sign, it's discarded by RIGHT$.

Now we have a string consisting only of a number. We next use VAL to convert the string to a number, which is assigned to the number variable *price*. (Remember: BASIC will treat the number variable *price* as a completely different variable than the string variable *price$*.) Now we can use the number in arithmetic formulas, as shown in the last line where we calculate a 6-percent tax using the converted price.



```
INPUT "Price"; price$
length = LEN(price$)
IF LEFT$(price$,1) = "$" THEN price$ = RIGHT$(price$, length - 1)
price = VAL(price$)
PRINT "Price ="; price
PRINT "Tax ="; price * .06
```

```
Price? $123.95
Price =  123.95
Tax =  7.437
```

FIGURE 10-14

Using the VAL function

Comparing strings

Notice that we are comparing two strings, using the equal sign, in the following line:

```
IF LEFT$(price$,1) = "$" THEN price$ = RIGHT$(price$, length - 1)
```

This works just the way comparing numbers does, returning a true value if the strings on both sides of the equal sign are equal.

The STR$ function

STR$, which we won't show in an example, is the opposite of VAL. It changes a numeric variable—the number 1099.5, for example—to a string that consists of the characters "1099.5".

SEARCHING FOR STRINGS: THE INSTR FUNCTION

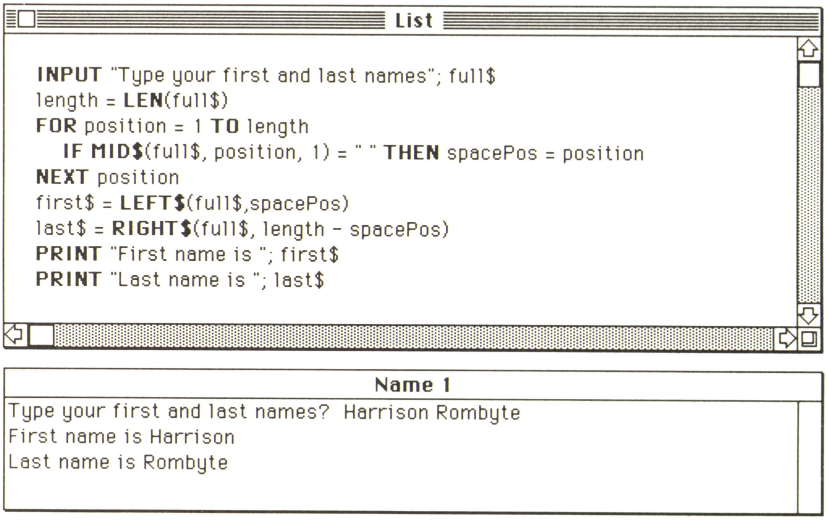
One of the most powerful string functions is INSTR. This function permits our program to examine a string to see if it contains another, shorter string within it. INSTR returns the position in the first string occupied by the second string. For example, the function INSTR("searching","ear") will return the value 2, since the *ear* string can be found starting at the second character in the *searching* string.

Searching for a single character

As an example of a useful role for INSTR, consider the following program. We want to ask the user to type both his or her first and last names, and then we want the program to figure out where the first name stops and the second name begins. The easy way to do this is to look for the space (" ") between the names.

Of course, we could look for this space using MID$. A program to do this is shown in Figure 10-15 on the next page. Notice that we must set up a loop, then go through the string one character at a time looking for the space. This approach can take a long time if a program must look through a lot of long strings.

Once the program finds where the space is, at position *spacePos,* it uses the LEFT$ and RIGHT$ functions to resolve the string *full$* into the set of characters before the space (the first name), and the set of characters after the space (the last name).



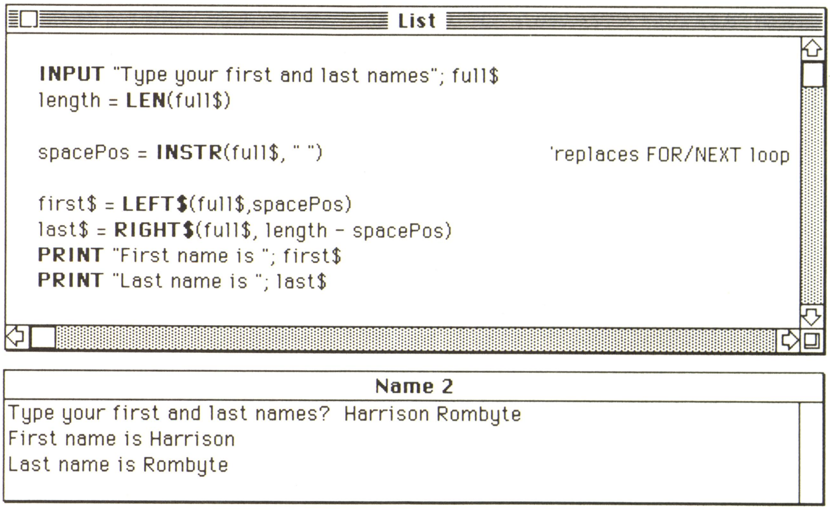
```
INPUT "Type your first and last names"; full$
length = LEN(full$)
FOR position = 1 TO length
   IF MID$(full$, position, 1) = " " THEN spacePos = position
NEXT position
first$ = LEFT$(full$,spacePos)
last$ = RIGHT$(full$, length - spacePos)
PRINT "First name is "; first$
PRINT "Last name is "; last$
```

```
Type your first and last names?  Harrison Rombyte
First name is Harrison
Last name is Rombyte
```

FIGURE 10-15

Searching for a character with MID$

Now look at the program in Figure 10-16. It performs a similar function, but does it more simply. It needs no loop because three instructions have been squeezed down to one: *spacePos = INSTR(full$, " ").* Also, because INSTR is a BASIC command, the operation is carried out much more quickly. In the example shown in Figure 10-16, INSTR assigns the value 9 to *spacePos,* since the space in *Harrison Rombyte* (the contents of *full$*) occupies the ninth position.



```
INPUT "Type your first and last names"; full$
length = LEN(full$)

spacePos = INSTR(full$, " ")                          'replaces FOR/NEXT loop

first$ = LEFT$(full$,spacePos)
last$ = RIGHT$(full$, length - spacePos)
PRINT "First name is "; first$
PRINT "Last name is "; last$
```

```
Type your first and last names?  Harrison Rombyte
First name is Harrison
Last name is Rombyte
```

FIGURE 10-16

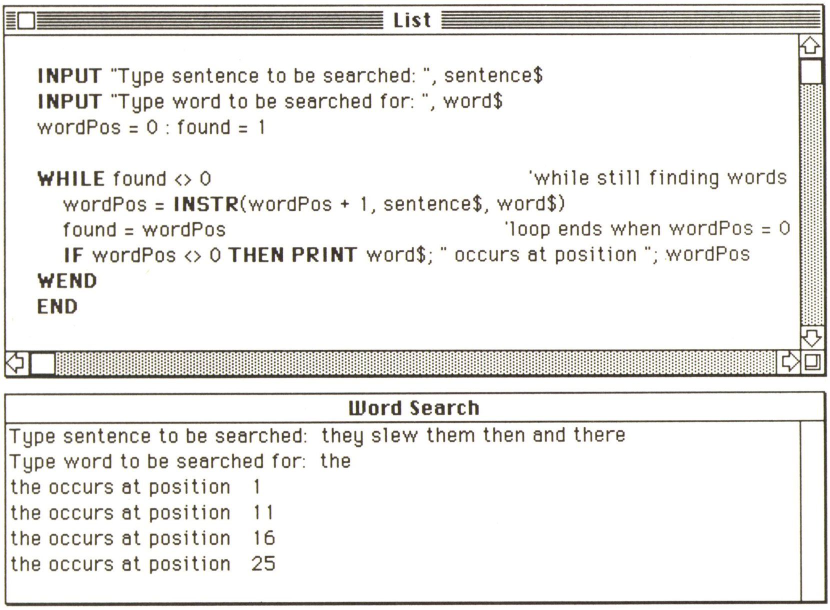
Searching for a character with INSTR

Searching for a string

The program shown in Figure 10-17 uses INSTR to search for a multi-character string instead of a single character as in the last example.

When run, this program prompts the user to type a sentence, and then type the word to be searched for within the sentence. The program looks for the first instance of the word in the sentence. In the example, it finds the word at position 1. However, we're not done yet. We want to find *all* the occurrences of the word in the sentence, not just the first one. To do this, we introduce a new parameter for INSTR: the location where the search should start. This is a number that precedes the string to be searched. For example, the function *INSTR(6,"concatenation","at")* will return the value 9, which is the starting position of the *second* occurrence of the string *at* in *concatenation.* It returns the second position because we told it to start searching at position 6, which is one character *past* the first occurrence of *at* at position 5.

Our program searches the sentence repeatedly for the occurrence of the word input into *word$.* The first time through the loop the program starts at position 1 (since we've set *wordPos* to 0 and the search starts at *wordPos + 1*). But when the program finds where the word first occurs, it changes *wordPos* to this number. Thus, the second time we go through the loop, the program starts searching one character *beyond* the position of the first match. This ensures that the program won't find the same match



```
INPUT "Type sentence to be searched: ", sentence$
INPUT "Type word to be searched for: ", word$
wordPos = 0 : found = 1

WHILE found <> 0                                'while still finding words
  wordPos = INSTR(wordPos + 1, sentence$, word$)
  found = wordPos                             'loop ends when wordPos = 0
  IF wordPos <> 0 THEN PRINT word$; " occurs at position "; wordPos
WEND
END
```

```
Type sentence to be searched:  they slew them then and there
Type word to be searched for:  the
the occurs at position   1
the occurs at position   11
the occurs at position   16
the occurs at position   25
```

FIGURE 10-17

Searching for a string with INSTR

over and over. It moves through the sentence, finding matches and printing them out, until it has found them all. At this point INSTR returns 0, *found* is set to 0, and the program ends.

THE LINE INPUT STATEMENT

INPUT is a very useful statement, as we've seen throughout this book. But even so, it has a drawback: The user can't include commas or quote marks in the string being entered, because if they are included, the message "?Redo from start" appears. This isn't good, because we can't guarantee what the user will type in. We want our programs to be able to deal with whatever characters the user types without producing this enigmatic error message (which has probably confused more people in the history of computing than any other phrase).

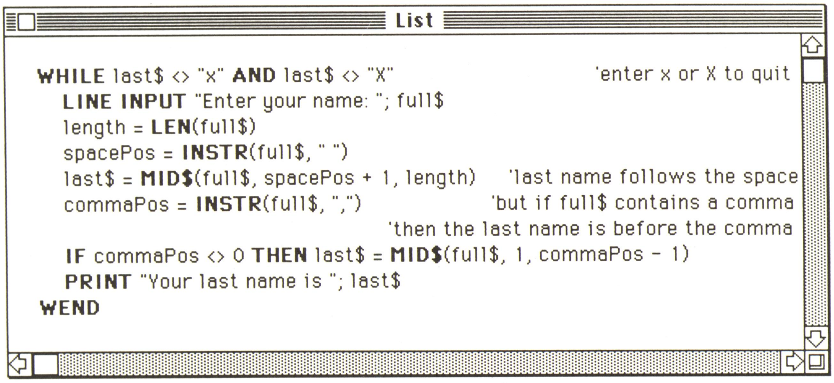
What we need is a statement that lets the user input anything at all: commas, numbers, letters, quotes, and so on. The statement that does this is LINE INPUT (yes, the space between the two words is necessary). This statement works much like INPUT, except that it accepts *everything* that's typed. As with INPUT, we can use a prompt string in quotes; however, LINE INPUT will not add a question mark to the prompt string.

The program shown in Figure 10-18 shows a situation where LINE INPUT is useful. In this program we ask the user to type a name. We expect the order to be first name followed by last name. But what if a user decides to type the last name, a comma, and then the first name? How can the program deal with this and figure it out? If we use INPUT and the user includes a comma, we'll get the "?Redo from start" error message. So we use LINE INPUT instead.

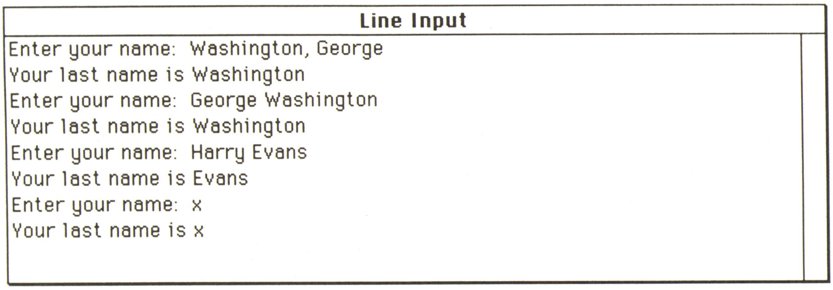
The program first uses INSTR to find where the space between the names is, and assigns this location in the string to *spacePos,* as in the previous program. It then makes the assumption that the name order is first name/last name, and assigns the characters *following the space* to the variable *last$.* Then it looks for a comma. If it finds one, it changes its mind about which name is last, and assigns the characters *preceding the comma* to the variable *last$.* It then prints out the last name.

FIGURE 10-18

Using LINE INPUT



```
WHILE last$ <> "x" AND last$ <> "X"                    'enter x or X to quit
  LINE INPUT "Enter your name: "; full$
  length = LEN(full$)
  spacePos = INSTR(full$, " ")
  last$ = MID$(full$, spacePos + 1, length)     'last name follows the space
  commaPos = INSTR(full$, ",")                 'but if full$ contains a comma
                                           'then the last name is before the comma
  IF commaPos <> 0 THEN last$ = MID$(full$, 1, commaPos - 1)
  PRINT "Your last name is "; last$
WEND
```



```
Enter your name:  Washington, George
Your last name is Washington
Enter your name:  George Washington
Your last name is Washington
Enter your name:  Harry Evans
Your last name is Evans
Enter your name:  x
Your last name is x
```

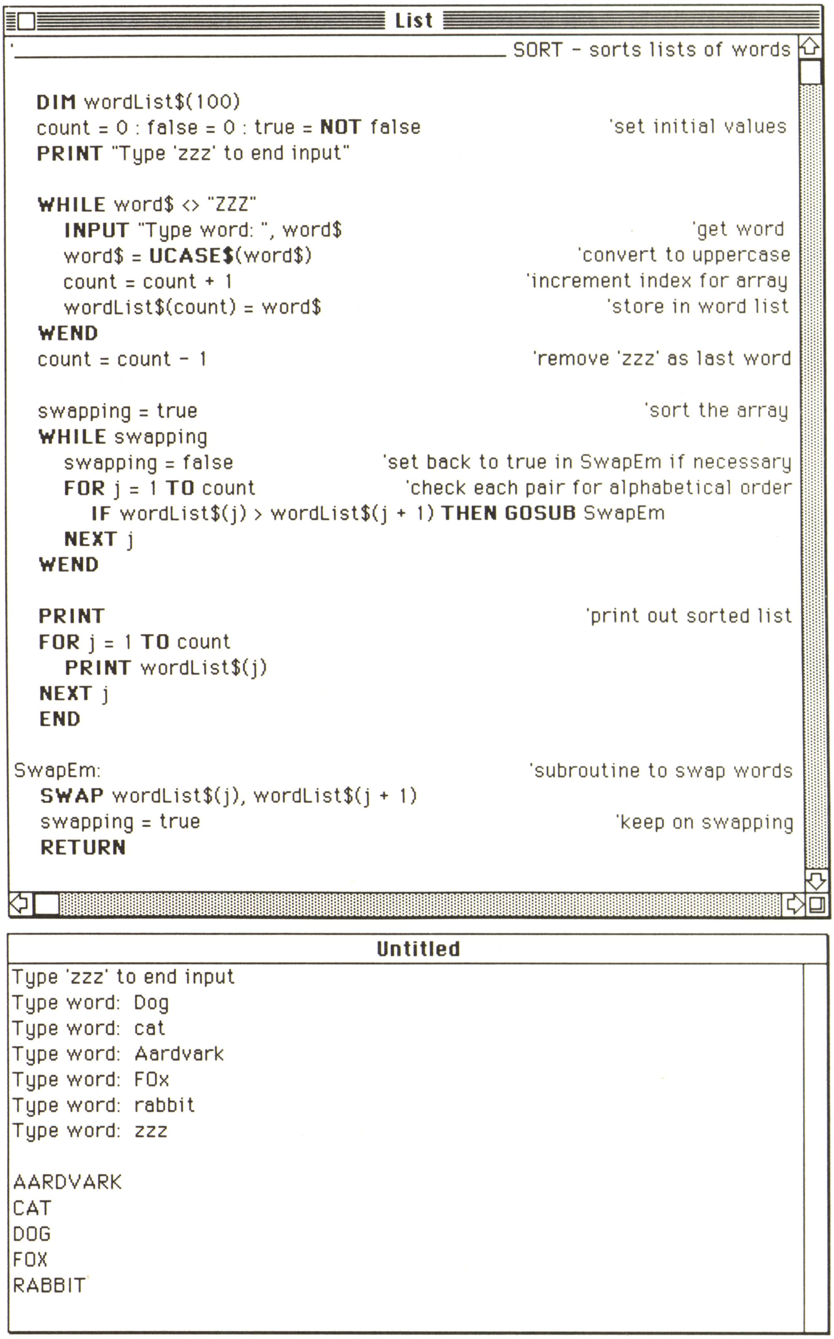
COMPARING STRINGS: SORTING

We saw earlier that we could compare two strings to see if they were identical by using the equal sign. It's also possible to compare strings using the logical operators > and <. But what does it mean to say that one string is larger or smaller than another? As you might guess, BASIC, in order to figure out how "large" a string is, compares the ASCII values of the individual characters in the string, starting with the first character. *A* is less than *B*, because 65, the ASCII value for *A*, is less than 66, the ASCII value for *B*. Also, *AA* is less than *AB*, since the first letters of the two strings are the same, but the second letters aren't, and *A* is again less than *B*.

The ability to compare strings in this way makes it possible to perform a variety of clever string operations, the most useful of which is probably to sort a list of strings into alphabetical order. The program shown in Figure 10-19, on the next page, accepts up to 100 words that the user types in, sorts them into alphabetical order, and then prints out the ordered list on the screen.

FIGURE 10-19

A sort program



```
'______________________________________ SORT - sorts lists of words

  DIM wordList$(100)
  count = 0 : false = 0 : true = NOT false              'set initial values
  PRINT "Type 'zzz' to end input"

  WHILE word$ <> "ZZZ"
    INPUT "Type word: ", word$                                'get word
    word$ = UCASE$(word$)                        'convert to uppercase
    count = count + 1                        'increment index for array
    wordList$(count) = word$                       'store in word list
  WEND
  count = count - 1                          'remove 'zzz' as last word

  swapping = true                                     'sort the array
  WHILE swapping
    swapping = false          'set back to true in SwapEm if necessary
    FOR j = 1 TO count         'check each pair for alphabetical order
      IF wordList$(j) > wordList$(j + 1) THEN GOSUB SwapEm
    NEXT j
  WEND

  PRINT                                          'print out sorted list
  FOR j = 1 TO count
    PRINT wordList$(j)
  NEXT j
  END

SwapEm:                                     'subroutine to swap words
  SWAP wordList$(j), wordList$(j + 1)
  swapping = true                                    'keep on swapping
  RETURN
```

```
Type 'zzz' to end input
Type word:  Dog
Type word:  cat
Type word:  Aardvark
Type word:  FOx
Type word:  rabbit
Type word:  zzz

AARDVARK
CAT
DOG
FOX
RABBIT
```

After setting up an array with 101 elements, setting the variable *count* (which counts how many words have been typed in) to 0, and telling the user that the last entry must be *zzz* to terminate the input list, the program goes into a loop to get the list of words from the user. Each word is placed in a different element of the array, starting sequentially with 0.

The UCASE$ function

Something to consider at this point is what will happen to our sorting process if the user types in some words with uppercase letters and some with lowercase? We don't want the alphabetizing process to pay any attention to whether the words consist of upper- or lowercase, but we know that the ASCII codes for upper- and lowercase are completely different. So to avoid problems, we'll turn all the words the user types into uppercase using the UCASE$ function.

Recall that the program in Figure 10-12 required a FOR...NEXT loop to change uppercase letters to lowercase. Changing lowercase letters to uppercase, on the other hand, requires only one function: UCASE$.

The UCASE$ function goes through a string character by character, and if a character is lowercase it changes it to uppercase. Numbers, punctuation, and other characters are left untouched.

The "bubble sort"

The "bubble sort" used in this program is not the fastest possible sorting method, but it's one of the easiest to explain. The program starts at the top of the list and compares two words. If they're out of order—that is, if the second one should come before the first alphabetically—then it swaps them in the array. It works its way down to the bottom of the list checking every pair in the array. Then it does the same thing again. Gradually the list becomes more and more ordered, as the words starting with *A,* for example, work their way up from the bottom, with repeated swaps, toward the top of the list. (This is why it's called a "bubble" sort: The words "bubble up" through the list.)

How does the program know when to stop? If it goes through the list and doesn't need to switch anything, that means that the list is in order and the process is over. In order to know when this has happened, the program keeps track of a variable called *swapping.* It sets this to false each time before going through the loop, and if it needs to make any switches, it sets it to true in the *SwapEm* subroutine, where the actual interchange of two elements is carried out. If no interchanges take place, *swapping* is never set to true, the program exits the WHILE...WEND loop, and the process is over. The PRINT section of the program then prints out the array elements in order.

The SWAP statement

You've probably noticed a new keyword, SWAP, in the *SwapEm* subroutine, in the line:

```
SWAP wordList$(j), wordList$(j + 1)
```

The purpose of the SWAP statement is to *exchange* the values of the two variables specified. All BASIC requires is that the two variables be of the same type (string or number), and that they both be already defined before the SWAP statement is encountered.

So in our example, if it's *true* in the main program that the second word in the pair (the word in *wordList$(j + 1)*) should precede the first in the pair (the word in *wordList$(j)*), the SWAP statement exchanges the contents of the two variables.

OTHER STRING FUNCTIONS

There are several string functions that make it easier to format printed output. We'll mention them briefly here.

The STRING$ function

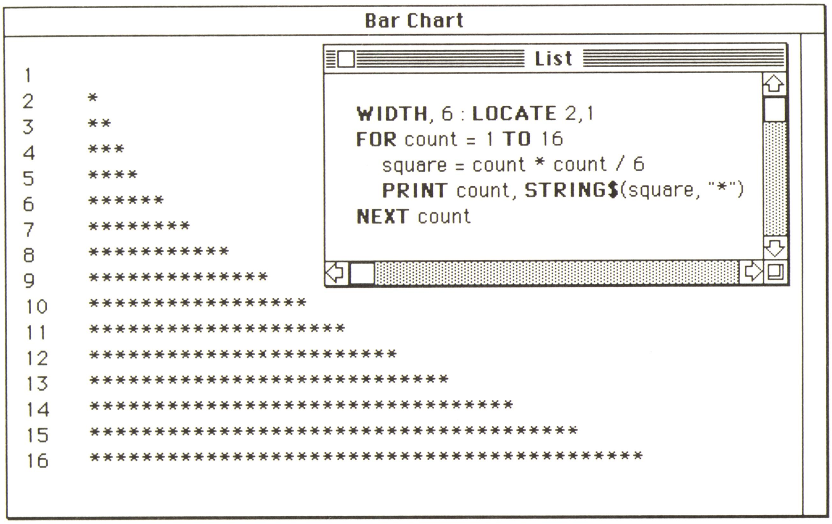
In creating forms and other documents you'll often want to print a whole row of some kind of character—asterisks or dashes, for example. You can do this by enclosing the string within quote marks, as in the statement *PRINT "--------------------"*. However, this can make your program listing look awkward, and waste a lot of your time typing the same character over and over again.

The STRING$ function avoids this problem. Assuming that the above statement contained 20 dashes, we could replace it with the statement *PRINT STRING$(20," – ")*. STRING$ is a function that returns a string consisting of a number of identical characters. The number, and the character to be repeated, are parameters of the function, as can be seen in the preceding example.

The STRING$ function can also be used in somewhat more creative ways. The program shown in Figure 10-20 uses STRING$ to produce a graph. In this case, what is graphed are the squares of the numbers from 1 to 16, represented with asterisks. The program calculates the square, and assigns it to the variable *square.* However, if we use this value, we'll find that most of the lines of asterisks don't fit on the screen. So we divide the number by 6 before assigning it to *square.* (The effect of this division is to *scale* the displayed output). The resulting number is used to tell STRING$ how many asterisks to print each time through the loop.

FIGURE 10-20

A program using the STRING$ function



```
WIDTH, 6 : LOCATE 2,1
FOR count = 1 TO 16
   square = count * count / 6
   PRINT count, STRING$(square, "*")
NEXT count
```

```
1
2     *
3     **
4     ***
5     ****
6     ******
7     ********
8     ***********
9     **************
10    *****************
11    ********************
12    ************************
13    ****************************
14    *********************************
15    **************************************
16    *******************************************
```

A similar method could be used to generate quick and easy graphs of any kind of data: income, sales information, or whatever.

The SPACE$ and SPC functions

The STRING$ function lets you print any number of a specific character you want. SPACE$ and SPC are more specialized: They are used to generate a fixed number of *spaces*. The statement *PRINT "X"; SPACE$(10); "X"* will thus print two *X*s separated by ten spaces.

Like STRING$, SPACE$ is a function that returns a string value. This value can be printed directly, as shown above, or assigned to a string variable for later use, as in the expression *tenSpaces$ = SPACE$(10)*.

The SPC function performs a similar function to SPACE$, but must be used in a PRINT statement; it does not return a value. For example, *PRINT "X"; SPC(10); "X"* generates two *X*s separated by ten spaces as in the last example, but SPC(10) doesn't return a string that can be assigned to a variable.

REFRESH-YOURSELF WINDOW

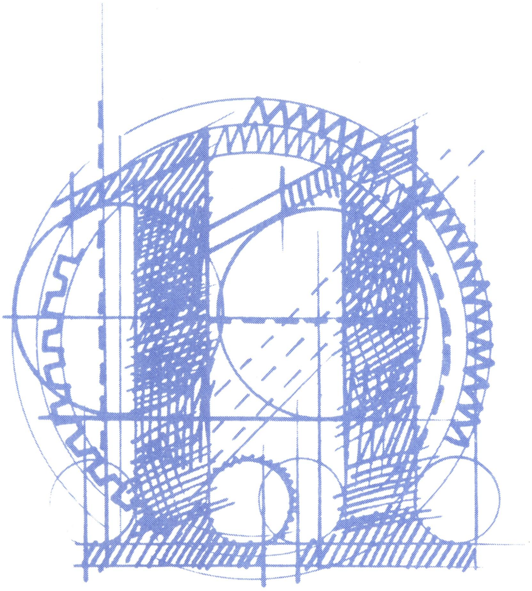
In this chapter we've learned the fundamentals of dealing with strings in our programs: what strings are, how to take them apart with the RIGHT$, LEFT$, and MID$ functions, and how to put them together with a + sign. We've also learned how characters are represented in the computer using

ASCII codes, and how the ASC, CHR$, VAL, and STR$ functions are used to convert back and forth between characters and ASCII codes.

We discovered the MID$ statement, and learned that it is different from the MID$ function; it is used to insert characters into an existing string. We found several ways to use the INSTR function, which searches for a character or string embedded in another string. We learned that two strings can be compared, and that the one higher in ASCII values comes later alphabetically, so that we can alphabetize lists of words. Finally we reviewed some functions that make it easier to format printed output.

SECTION II

Advanced BASIC for the Macintosh



The Mouse and Event Trapping

In Chapter 6, "Loops," we learned how our programs could find out if the mouse button had been pushed, and where the mouse pointer was on the screen. However, the mouse can do much more. In this chapter we are going to concentrate on the mouse and its capabilities. We'll learn how to tell when the mouse has been clicked once, twice, or three times, and what we can do with this information. We'll also learn how our program can tell when the mouse pointer has been dragged across the screen, and where the drag began and ended.

Besides programming with the mouse using the WHILE...WEND loop, as we've done, we can also program with something called *event traps.* This leads to a different way to structure our programs. We'll learn how our programs can use event trapping, and why this is a powerful programming technique.

Finally, we'll finish off the chapter by discovering INKEY$, a command that lets us input letters and numbers into a program in much the same way that the mouse functions let us input information about the mouse's pointer position and button status.

MOUSE BUTTON STATUS

We already know that if the MOUSE(0) function returns the value 1, it means that the mouse button has been pushed, while a value of 0 indicates that the button has not been pushed. However, MOUSE(0) can return a variety of values besides 0 and 1. These values range from −3 to 3, and are summarized in the table shown in Figure 11-1. To get a feeling for what these values mean, type in and run the program shown in Figure 11-2.

While the program runs, try clicking the mouse button once: A 1 is printed on the screen. Click it twice in rapid succession, and a 2 is printed out. Three times gives you a 3. If you press the button and hold it down, a −1 is printed. If you click twice, but hold the button down after the second press, the program returns a −2. Similarly, clicking three times but leaving the button down after the last click elicits a −3.

We now have six possible pieces of information that the MOUSE(0) function can return. We'll show you a program that uses three of these possible values, but before we get to that we should examine some of the other features of our program.

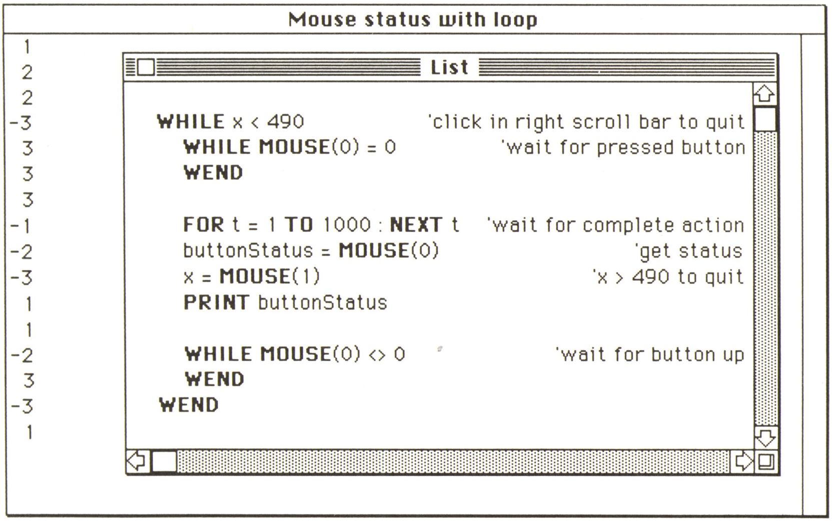
FIGURE 11-1

The possible MOUSE(0) values

	Button released after last click	Button held down after last click
Single-click	1	−1
Double-click	2	−2
Triple-click	3	−3

FIGURE 11-2

A program that prints the status of the mouse button



```
1
2
2
-3
3
3
3
-1
-2
-3
1
1
-2
3
-3
1
```

```
WHILE x < 490                 'click in right scroll bar to quit
   WHILE MOUSE(0) = 0                  'wait for pressed button
   WEND

   FOR t = 1 TO 1000 : NEXT t  'wait for complete action
   buttonStatus = MOUSE(0)                          'get status
   x = MOUSE(1)                                 'x > 490 to quit
   PRINT buttonStatus

   WHILE MOUSE(0) <> 0                       'wait for button up
   WEND
WEND
```

Time delays and mouse-human interaction

The Macintosh is a very fast computer. In the time it takes you to click the mouse button, it can perform hundreds of BASIC instructions. This disparity in speed between the human user and the computer can lead to problems. When we're programming with the mouse, we need to keep in mind what the program is doing and what the human button-presser is doing at each moment, and we need to make sure that they don't get in each other's way.

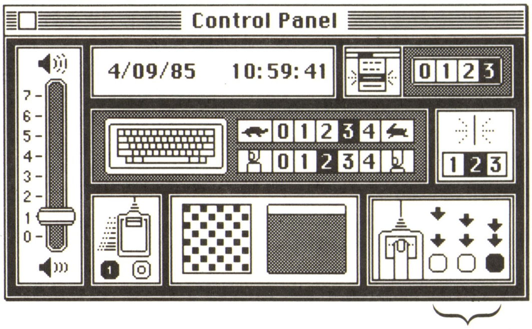
In our program there are three places where we build in different kinds of delays to avoid these mouse-human interaction problems. The heart of the program is the two lines *buttonStatus = MOUSE(0)* and *PRINT buttonStatus.* However, we can't just plunge into these statements. First, we don't want to print anything if the button has not been pressed, so we wait in the first WHILE...WEND loop for this to happen. As soon as the button has been pressed, we fall out of this loop.

However, when the program first finds out that the button has been pressed, that's *all* it knows: It doesn't know how many times it was pressed, or whether it was left up or down. The program is ignorant of these factors at this point because not enough time has gone by for the user to complete these actions. So the program must *wait* for the user to have time to finish pressing the button as many as three times. That's the purpose of the program line:

```
FOR t = 1 TO 1000 : NEXT t
```

FIGURE 11-3

Changing the mouse button rate in the Control Panel



Click here to change mouse button response time

This line delays the program just long enough for up to three repetitive clicks. Then, and only then, does the program actually take the number that corresponds to the button status and print it out.

Finally, if the program's user holds down the button to elicit a minus number from the program, we must wait until the button is then *released* before we go back to the beginning of the program loop. Otherwise, we'll dive right into the whole process again, and record the release of the mouse button as the beginning of an entirely new sequence of button actions, rather than as the end of the last one. This is the purpose of the last WHILE...WEND loop.

These three delays allow us to obtain complete and accurate information about the mouse's button status. (When we get to event trapping later in this chapter you will see how this time-delay process can be greatly simplified.)

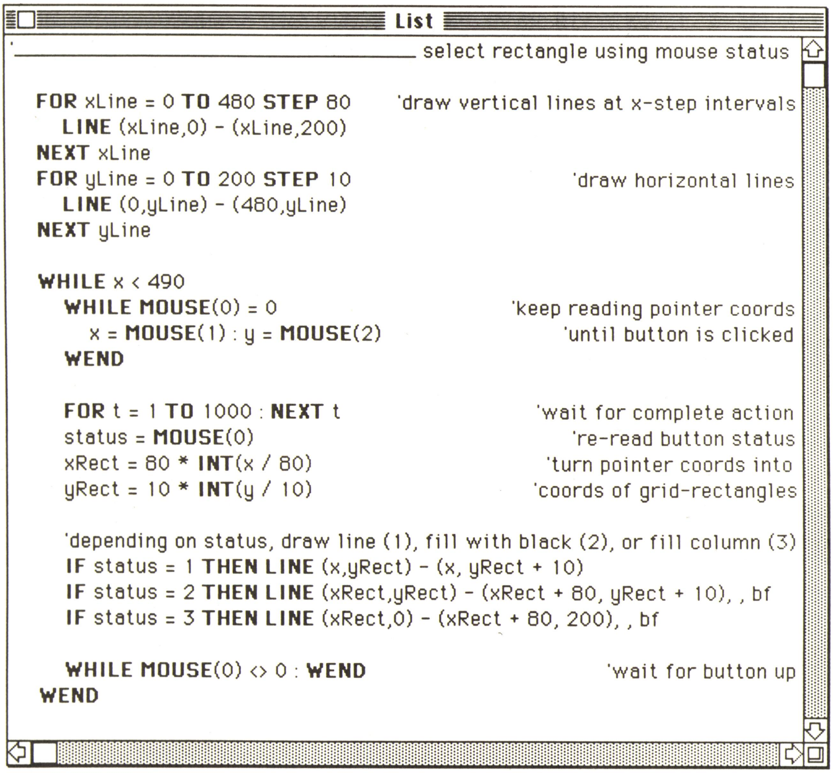
If you play with the program you might find that sometimes a number other than the right number of clicks will be displayed. This happens because the operation of the mouse in Microsoft BASIC is affected by the settings found in the lower right corner of the *Control Panel,* as shown in Figure 11-3.

The three choices found there govern the acceptable interval between successive clicks, and whether those clicks are interpreted as two single-clicks or one double-click. Selecting the rightmost button as shown in the figure means that two clicks must be closer together in order to be accepted as a double-click. Try experimenting with these settings to determine their effect on the operation of the program.

The Select-Rectangle program

As an example of the versatility of the MOUSE(0) function, enter and run the program in Figure 11-4. This program first draws a grid of vertical and horizontal lines on the screen. Once the lines are drawn, we can use the mouse to cause different actions in the grid. A single-click draws a vertical line in a particular "cell" at the position of the mouse pointer. A double-click "selects" an entire cell, causing it to turn black. A triple-click "selects" an entire column, causing it to turn black, as happened with the fifth column in the figure.

Although this program accomplishes nothing useful in itself, it's easy to see how similar mouse techniques could be used in a spreadsheet, word processor, or similar type of program where a variety of different actions need to be performed at particular locations on the screen.



```
'________________________________________ select rectangle using mouse status

 FOR xLine = 0 TO 480 STEP 80          'draw vertical lines at x-step intervals
   LINE (xLine,0) - (xLine,200)
 NEXT xLine
 FOR yLine = 0 TO 200 STEP 10                                'draw horizontal lines
   LINE (0,yLine) - (480,yLine)
 NEXT yLine

 WHILE x < 490
   WHILE MOUSE(0) = 0                                     'keep reading pointer coords
     x = MOUSE(1) : y = MOUSE(2)                              'until button is clicked
   WEND

   FOR t = 1 TO 1000 : NEXT t                                'wait for complete action
   status = MOUSE(0)                                           're-read button status
   xRect = 80 * INT(x / 80)                                   'turn pointer coords into
   yRect = 10 * INT(y / 10)                                  'coords of grid-rectangles

   'depending on status, draw line (1), fill with black (2), or fill column (3)
   IF status = 1 THEN LINE (x,yRect) - (x, yRect + 10)
   IF status = 2 THEN LINE (xRect,yRect) - (xRect + 80, yRect + 10), , bf
   IF status = 3 THEN LINE (xRect,0) - (xRect + 80, 200), , bf

   WHILE MOUSE(0) <> 0 : WEND                                      'wait for button up
 WEND
```

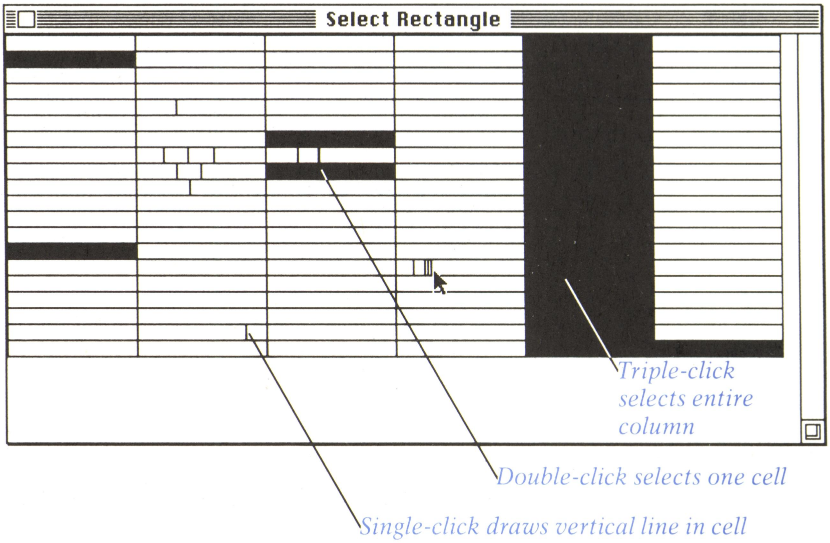
FIGURE 11-4

The listing and output of the Select-Rectangle program

(continued)

FIGURE 11-4

(continued)



Triple-click selects entire column

Double-click selects one cell

Single-click draws vertical line in cell

DRAGGING THE MOUSE

You are already familiar with the technique of dragging the mouse pointer across the screen. You drag the pointer across a word or a complete line in your BASIC program so you can cut the word or line or copy it to another part of your program. You do the same thing in MacWrite. And, if you're familiar with MacPaint, you know that you can create rectangles and other shapes, as well as selecting rectangular portions of the screen, by dragging the mouse pointer with the appropriate tools selected.

Is it possible to use mouse dragging in our own programs? It is: The MOUSE functions give our program the ability to learn everything it needs to know about a mouse drag.

So far we've learned about MOUSE(0), which gives us the button status, and MOUSE(1) and MOUSE(2), which return the x and y coordinates of the mouse pointer. However, there are four other MOUSE functions, and these four functions make it easy for our programs to work with mouse dragging.

To drag the mouse, you move the pointer to a specific location, push the button down, move the pointer to another location, and release the button. The dragging operation thus generates four pieces of information: the x and y coordinates where the button was pushed down, and the x and y coordinates where it was released. These four pieces of information are available in the functions MOUSE(3), MOUSE(4), MOUSE(5), and MOUSE(6), as shown in the table in Figure 11-5.

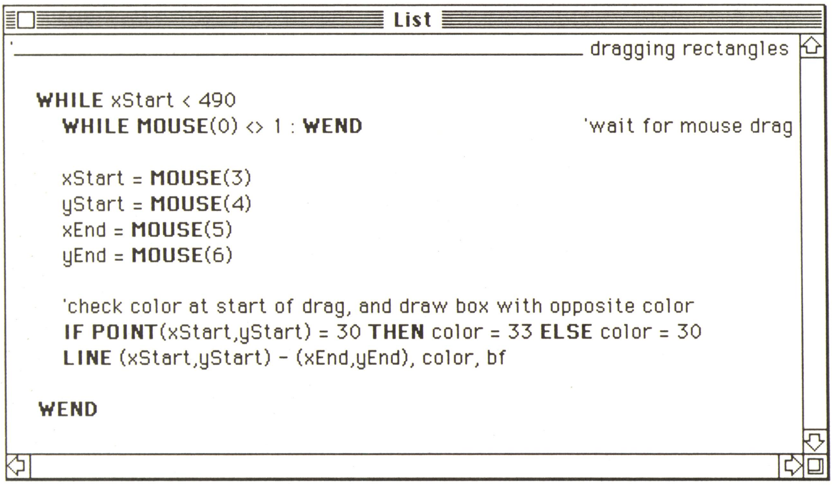
FIGURE 11-5

The mouse-drag functions

Function	Information Reported
MOUSE(3)	Starting x coordinate
MOUSE(4)	Starting y coordinate
MOUSE(5)	Ending x coordinate
MOUSE(6)	Ending y coordinate

The example program listed in Figure 11-6 uses these functions to record a mouse drag and generate a filled box whose corners coincide with the starting and ending points of the drag. The program can generate two kinds of rectangles: black and white. If the drag is started in a white area of the screen, the program generates a black rectangle; if the drag is started in a black area, the program generates a white rectangle.

The outer WHILE...WEND loop allows the user to exit from the program by clicking the mouse button while the pointer is in the vertical scroll bar—a technique we have used in previous chapters. The inner WHILE...WEND loop waits for a mouse drag to occur. MOUSE(0) returns a value of 1 only if the button has been pushed *and released*, which doesn't happen *during* a drag operation. So as soon as the button is *released*, the program exits from the loop and reads the four pieces of information about the starting and ending coordinates of the drag. These values are assigned to the variables *xStart*, *yStart*, *xEnd*, and *yEnd*. Using these coordinates, the program then executes the LINE statement to draw a rectangle whose upper left corner is at coordinates *(xStart,yStart)*, and whose lower right corner is at coordinates *(xEnd,yEnd)*, as shown in the Output window of Figure 11-6.



```
'_______________________________________________ dragging rectangles

  WHILE xStart < 490
     WHILE MOUSE(0) <> 1 : WEND                       'wait for mouse drag

     xStart = MOUSE(3)
     yStart = MOUSE(4)
     xEnd = MOUSE(5)
     yEnd = MOUSE(6)

     'check color at start of drag, and draw box with opposite color
     IF POINT(xStart,yStart) = 30 THEN color = 33 ELSE color = 30
     LINE (xStart,yStart) - (xEnd,yEnd), color, bf

  WEND
```

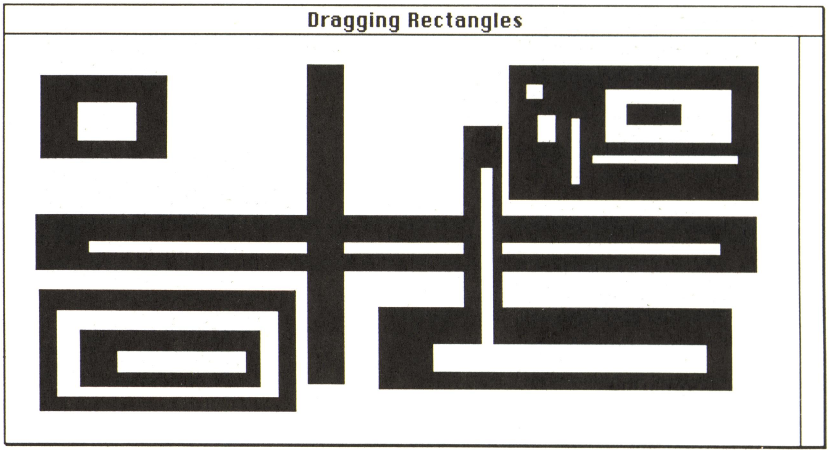
FIGURE 11-6

The listing and output of the Dragging-Rectangles program

(continued)

FIGURE 11-6

(continued)



How does the program decide whether to draw a white rectangle or a black one? Black and white are defined as *colors* in Microsoft BASIC. Black has a value of 33, and white has a value of 30, as we learned in Chapter 4, "Introducing Graphics." To know what color rectangle to draw, the program must determine the color already existing at a particular point on the screen. This is handled by the POINT function.

The POINT function

The POINT function must be given two pieces of information: the x coordinate and the y coordinate of a particular pixel on the screen. It then returns a value indicating the color of the pixel it finds there: 30 for white, or 33 for black.

Depending upon what the POINT function discovers, the IF... THEN... ELSE statement in the program then changes the variable *color* to the opposite value. Thus when the LINE statement is executed, it draws a filled box of the color opposite to whatever the screen color was at the starting point of the drag. Note that although we can use either 0 or 30 for white and 1 or 33 for black in the LINE statement, POINT only returns the value 30 or 33.

Drawing while dragging

The problem with our current program is that it doesn't draw anything while the mouse is being dragged. There is no visual indication of what the dragging is accomplishing: The user is "flying blind" until after the drag is completed and the rectangle is drawn. To be useful, a program should show an outline of a rectangle, or highlight text, or do something similar during the dragging process so the user can see what's going on.

The program listed in Figure 11-7 does exactly this, creating rectangles that are drawn and erased one after the other until the button is released, and then filling the final rectangle with the appropriate color.

For simplicity, this is a one-shot program: It creates a single rectangle. However, you could easily put it into a loop similar to that in the last program, to create a number of rectangles instead of just one.

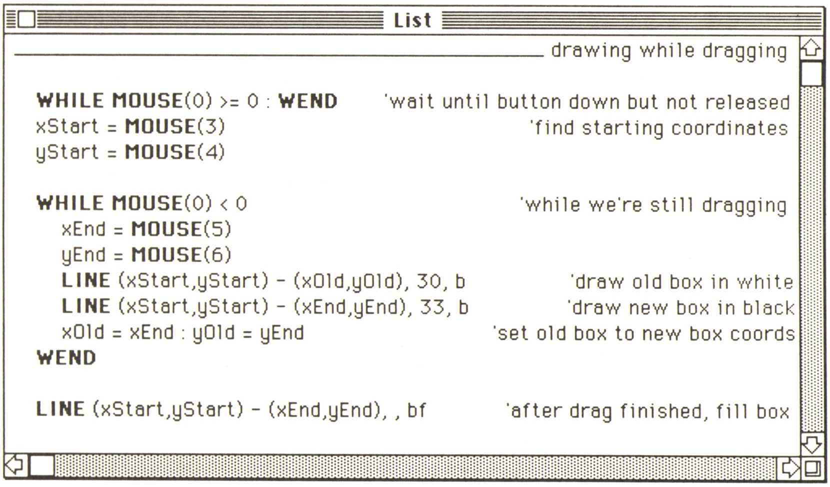
The program first waits for the button to be pressed, but not released, which means the mouse is being dragged. When this happens, MOUSE(0) is negative, and the program falls out of the first WHILE... WEND loop. The dragging process has begun.

Next, the program records the starting point of the drag: the location of the mouse pointer when the button was pressed. As we learned in the last program, this information is available with the MOUSE(3) and MOUSE(4) functions. These coordinates, which we assign to the variables *xStart* and *yStart*, remain unchanged for the remainder of the program.

Next, the program enters a loop where it cycles until the mouse button is released. In this loop the program records, in the *xEnd* and *yEnd* variables, the current end points of the dragging operation, which it finds from MOUSE(5) and MOUSE(6). These points are constantly changing as the mouse moves.

FIGURE 11-7

Rectangles that change as the user drags the mouse

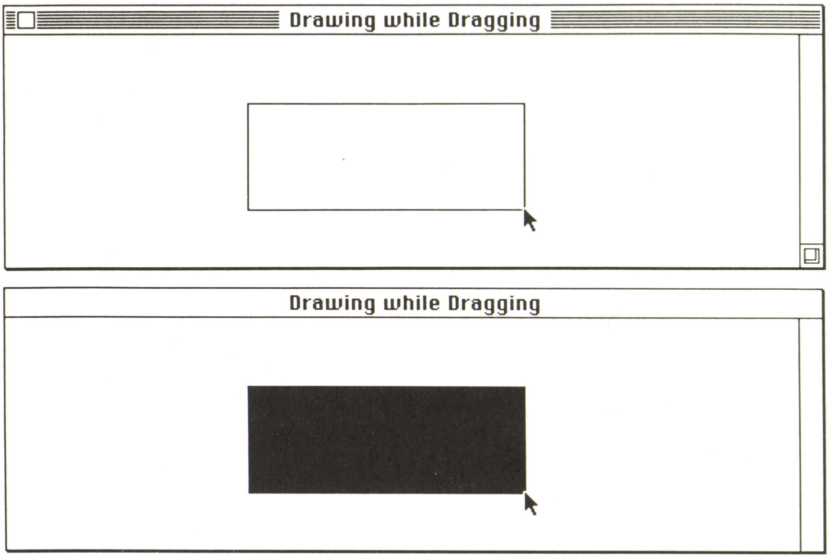


```
_____________________________________________ drawing while dragging

WHILE MOUSE(0) >= 0 : WEND        'wait until button down but not released
xStart = MOUSE(3)                            'find starting coordinates
yStart = MOUSE(4)

WHILE MOUSE(0) < 0                          'while we're still dragging
   xEnd = MOUSE(5)
   yEnd = MOUSE(6)
   LINE (xStart,yStart) - (xOld,yOld), 30, b        'draw old box in white
   LINE (xStart,yStart) - (xEnd,yEnd), 33, b        'draw new box in black
   xOld = xEnd : yOld = yEnd               'set old box to new box coords
WEND

LINE (xStart,yStart) - (xEnd,yEnd), , bf        'after drag finished, fill box
```

(continued)

FIGURE 11-7

(continued)



Once the program has stored the values for the pointer's starting and ending (x,y) coordinates, it uses them to draw an empty rectangle from the starting point to the current end point of the drag operation with the statement:

```
LINE (xStart,yStart) - (xEnd,yEnd), 33, b
```

This rectangle constantly expands (and possibly contracts) as the mouse pointer moves around on the screen. But as the size of the rectangle changes we want to display only the *current* rectangle; we don't want to accumulate a whole bunch of rectangles of various sizes on the screen at the same time. We therefore need a way to erase the last rectangle before drawing the new one.

To erase the old rectangle, the program must *remember* the coordinates of the rectangle's ending point, even though the mouse pointer is moving to a new location. It therefore records the information in variables *xOld* and *yOld*. The next time through the loop, after it has drawn the new rectangle, the program erases the old one with the statement:

```
LINE (xStart,yStart) - (xOld,yOld), 30, b
```

The result is a series of rectangles that gives the impression of a single rectangle that's rapidly changing its shape as dictated by the mouse pointer.

Finally, when the mouse button is released at the end of the drag, the program exits the WHILE...WEND loop and the last line draws a filled rectangle using the starting and ending points of the drag operation.

MOUSE STATUS AND EVENT TRAPPING

So far our programs in this chapter have had to repeatedly execute the MOUSE functions to find out if the button had been pressed, and if so how many times, and so forth. In other words, the program was devoting itself to finding out about the mouse, not unlike a small child standing around saying, "Have you pressed the button yet? Huh? Have you pressed the button yet? Have you? Huh?"

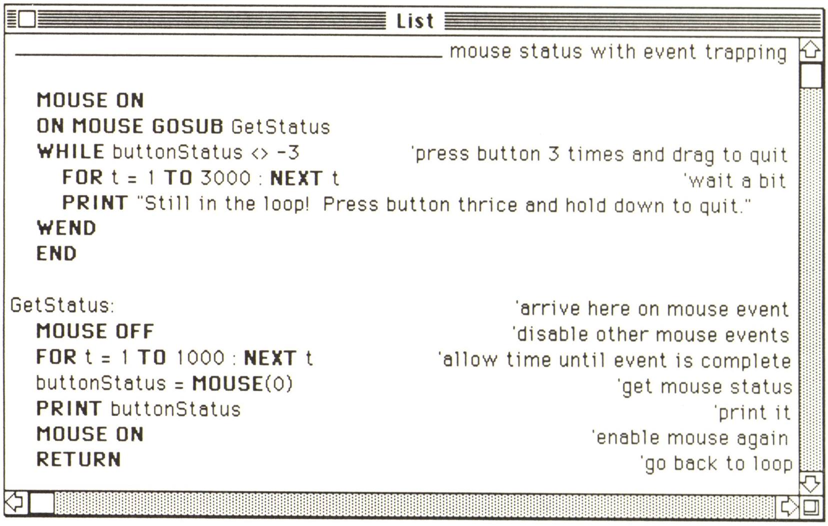
This is fine if our program has nothing better to do. However, we will often want a program to be processing other program statements and *at the same time* be waiting for the user to press the mouse button. For instance, in a spreadsheet program you may want the program to perform a complex calculation, but you don't want the user to have to wait until the calculation is completed before selecting another cell by clicking on it with the mouse. Or, in a game program, you may want to be constantly moving an object while allowing the program to instantly fire a weapon when the button is pressed. In cases like these, we need a way for the program to do one thing but at the same time be aware of, or be waiting for, something else, so it can be interrupted to deal with it.

This problem can be solved by the BASIC statement *ON MOUSE GOSUB.* The "something else" that the program is waiting for—in this case the mouse button being pressed—is called an *event,* and the technique of figuring out when such an event has occurred, and responding to it, is called *event trapping.*

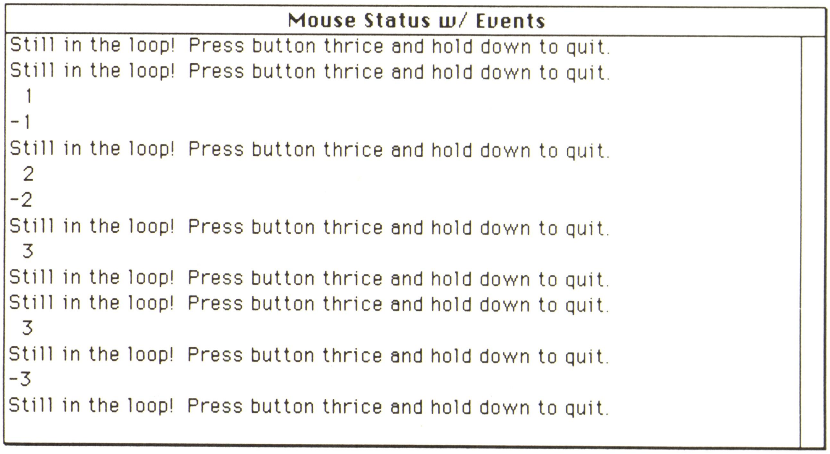
To get an idea how event trapping works, let's examine the program shown in Figure 11-8, on the following page. This program performs exactly the same function as the one at the beginning of this chapter: It prints out the mouse button status from −3 to 3. However, it does it in a different way. Instead of constantly monitoring the status of the mouse in a loop, as our first program did, it uses event trapping to discover if the mouse button has been pressed. Although the two programs look similar and do similar things, the difference between them is important. In our first example, the program is *actively* finding out what the mouse status is, all the time using the MOUSE(0) function. In the second example, the program acts as if it doesn't care about the mouse; it's "thinking" about something else. Yet, if the mouse button is pressed, it can still respond.

FIGURE 11-8

Finding out the mouse status with event trapping



```
  MOUSE ON
  ON MOUSE GOSUB GetStatus
  WHILE buttonStatus <> -3          'press button 3 times and drag to quit
    FOR t = 1 TO 3000 : NEXT t                                'wait a bit
    PRINT "Still in the loop!  Press button thrice and hold down to quit."
  WEND
  END

GetStatus:                                        'arrive here on mouse event
  MOUSE OFF                                       'disable other mouse events
  FOR t = 1 TO 1000 : NEXT t               'allow time until event is complete
  buttonStatus = MOUSE(0)                                 'get mouse status
  PRINT buttonStatus                                              'print it
  MOUSE ON                                              'enable mouse again
  RETURN                                                   'go back to loop
```



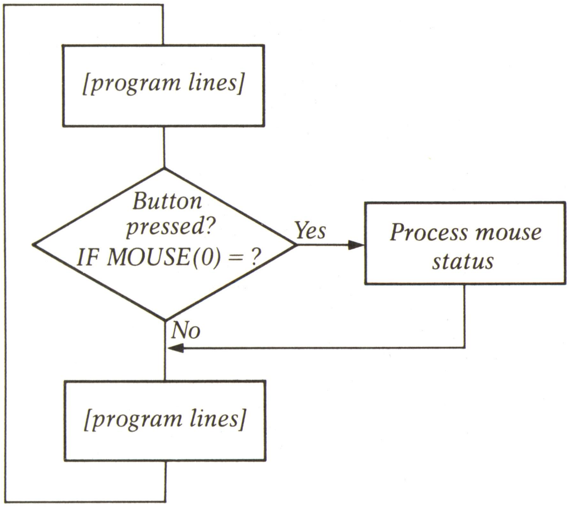
```
Still in the loop!  Press button thrice and hold down to quit.
Still in the loop!  Press button thrice and hold down to quit.
 1
-1
Still in the loop!  Press button thrice and hold down to quit.
 2
-2
Still in the loop!  Press button thrice and hold down to quit.
 3
Still in the loop!  Press button thrice and hold down to quit.
Still in the loop!  Press button thrice and hold down to quit.
 3
Still in the loop!  Press button thrice and hold down to quit.
-3
Still in the loop!  Press button thrice and hold down to quit.
```

The difference between the two programs is shown schematically in the two flowcharts of Figure 11-9. The first flowchart in this figure diagrams the loop approach to finding the mouse-button status. Notice that the program must check the mouse status each time through the loop. The second flowchart illustrates the event-trapping approach. After setting up the loop, the program doesn't need to check the mouse button at all. If the button is pressed, the program jumps out of the loop automatically, goes to the routine that processes the mouse status, then returns to the same point in the loop when this processing has been completed. BASIC checks the mouse status on its own, once every time a statement is executed. It is something like putting an *IF MOUSE(0) <> 0 THEN GOSUB process mouse* statement between every statement in the program.

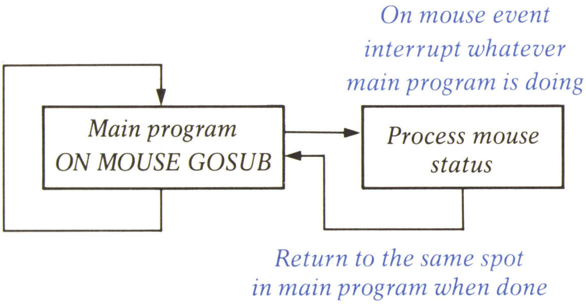
FIGURE 11-9

Flowcharts of two mouse-status programs

Looping approach


[program lines]
Button pressed? IF MOUSE(0) = ?
Yes
Process mouse status
No
[program lines]

Event-trapping approach


Main program ON MOUSE GOSUB
Process mouse status

On mouse event interrupt whatever main program is doing

Return to the same spot in main program when done

Anatomy of an event-trapping program

Let's return to our program in Figure 11-8 and look at the various elements in detail.

Most of the time, our program is doing something that has nothing to do with the mouse: printing "Still in the loop!" The program keeps executing this loop until a mouse event takes place.

But before our program goes off to process the loop, we must tell it two things. First, we must tell it, "Be prepared to respond if the mouse button is pressed." We must also tell it, "If the mouse button *is* pressed, here's what you do." Let's look at how we perform these two functions.

The MOUSE ON statement

MOUSE ON alerts BASIC that there may be a mouse event (a button press), and that if there is, something should be done about it, which means the program must jump to a subroutine set up to take care of the event. MOUSE ON *itself* doesn't tell BASIC *what* should be done, but simply makes BASIC aware of the possibility of a mouse event. If MOUSE ON is *not* included in the program, then nothing will happen when the mouse button is pressed, even if a subsequent statement tells BASIC what to do about it. (Of course, you could design your program to check the MOUSE(0) function with the same kind of loop described earlier.)

The ON MOUSE GOSUB statement

The ON MOUSE GOSUB statement tells BASIC what to do when the mouse button is pressed. More specifically, it tells BASIC *what subroutine to execute.* In our program, the subroutine is called *GetStatus.* Thus the statement:

```
ON MOUSE GOSUB GetStatus
```

means, "If the mouse button is pressed at any time (and the mouse function has been turned on with MOUSE ON), then execute the *GetStatus* subroutine."

The MOUSE OFF statement

As you can see in Figure 11-8, most of the *GetStatus* subroutine is similar to the original button-status program shown in Figure 11-2. The FOR...NEXT loop slows the program to allow the user to finish clicking the button, then reads the button status and prints it out. But what about the MOUSE OFF statement at the beginning of the routine, and the MOUSE ON statement at the end?

These two statements solve the following problem for us. Suppose our program found out that the mouse button was pressed and, using the ON MOUSE GOSUB statement, went off to deal with the button press. Then, while it was dealing with the first button press and was in the middle of the *GetStatus* subroutine, the button was pressed again. What would happen? Off it would go again to this routine, as directed by the *ON MOUSE GOSUB GetStatus* statement, but without ever returning from executing the routine the first time! The result of this would be total confusion for the program.

We avoid this problem neatly by *disabling* the possibility of any further mouse events while we're dealing with the first one. *MOUSE OFF* disables the mouse event trapping, in the same way that MOUSE ON enabled it at the beginning of the program. In fact, we use MOUSE ON again, at the end of our routine, to enable mouse event trapping again so the whole process can be repeated. This ensures that we won't try to process one mouse event before we've finished with the last one.

Notice that the MOUSE ON and MOUSE OFF statements enable us to do away with the WHILE...WEND button release loop at the end of the original program.

Working while we wait

Our previous example nicely illustrated when the program was in the loop and when it was sent to the *GetStatus* subroutine by the ON MOUSE GOSUB statement. Now let's try a program that does something a bit more visually interesting with the mouse-event statements. Type in and run the program listed on the next page in Figure 11-10.

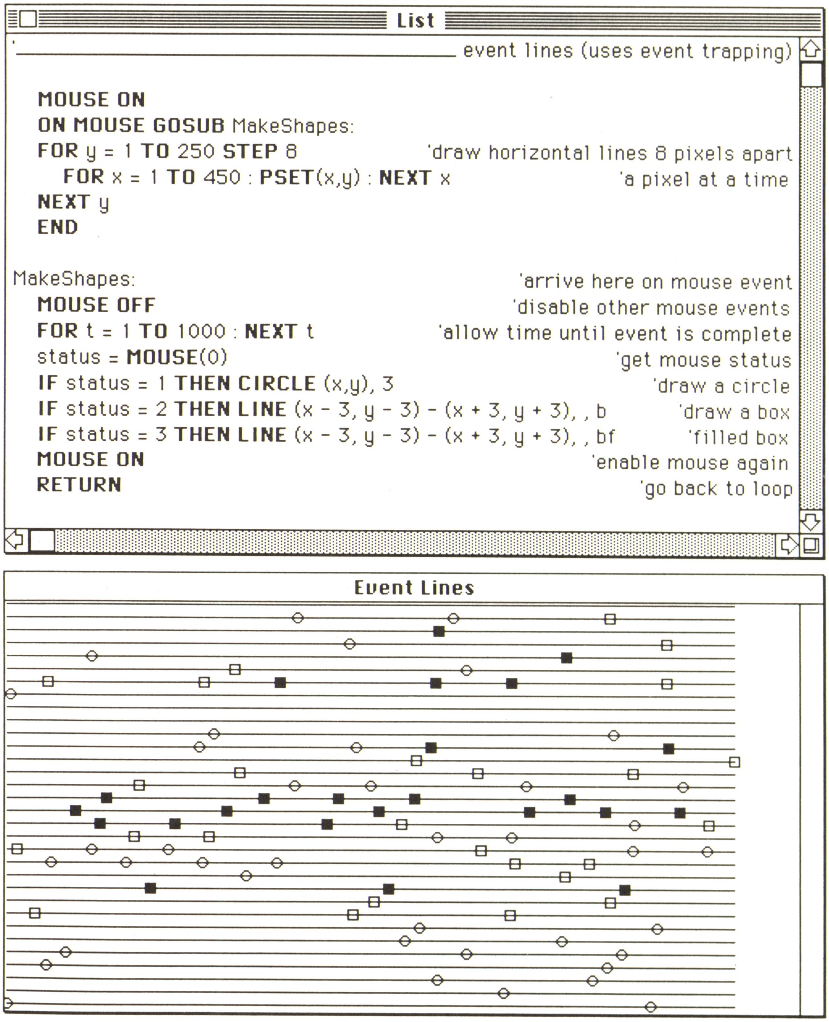
The "useful" work performed by this program is drawing a series of horizontal lines on the screen. This is accomplished by the nested FOR...NEXT loops at the beginning of the program. The PSET statement in the inner loop plots the line, pixel by pixel. (Of course, lines can be drawn faster with the LINE statement, but then things would go too fast to follow in this demonstration program.)

As the program draws lines, it's also waiting for a mouse event, since we have issued the *MOUSE ON* and *ON MOUSE GOSUB MakeShapes* statements at the beginning of the program. If the button is pressed, control is immediately transferred to the *MakeShapes* routine, which contains three possibilities. If the button has been pressed once, a circle is drawn at the current x and y coordinates (that is, at the current point in the line as drawn thus far). If the mouse was clicked twice, a box is drawn, and if it was clicked three times, a filled box is drawn. Examples of these possibilities, as generated by the mad clicker, are shown in the Output window of Figure 11-10.

Of course, in a real program, the work the program was doing would be more interesting than drawing lines. For example, it could be doing mathematical calculations, producing sounds, reading or writing information to the disk, or rearranging the words on the screen in a word-processing program.

FIGURE 11-10

The listing and output of the Event-Lines program



```
'_______________________________________ event lines (uses event trapping)

  MOUSE ON
  ON MOUSE GOSUB MakeShapes:
  FOR y = 1 TO 250 STEP 8              'draw horizontal lines 8 pixels apart
    FOR x = 1 TO 450 : PSET(x,y) : NEXT x                      'a pixel at a time
  NEXT y
  END

MakeShapes:                                          'arrive here on mouse event
  MOUSE OFF                                         'disable other mouse events
  FOR t = 1 TO 1000 : NEXT t                   'allow time until event is complete
  status = MOUSE(0)                                          'get mouse status
  IF status = 1 THEN CIRCLE (x,y), 3                          'draw a circle
  IF status = 2 THEN LINE (x - 3, y - 3) - (x + 3, y + 3), , b        'draw a box
  IF status = 3 THEN LINE (x - 3, y - 3) - (x + 3, y + 3), , bf       'filled box
  MOUSE ON                                                 'enable mouse again
  RETURN                                                    'go back to loop
```

OTHER KINDS OF EVENTS

The mouse is actually only one of several possible Macintosh events we can program. Besides ON MOUSE GOSUB, there are also the following possibilities in Microsoft BASIC:

- ON BREAK GOSUB
- ON DIALOG GOSUB
- ON MENU GOSUB
- ON TIMER GOSUB

Each of these statements can be used in much the same way as ON MOUSE GOSUB. For each one, there are corresponding statements to activate and deactivate the function. For instance, for the menu, there is MENU ON and MENU OFF, and so forth.

In later chapters you'll learn how to use some of these functions, such as menus and dialog boxes. As you read about these functions, remember that they can be used not only in loops, but as events as well. In fact, event trapping makes it possible to structure an entire program as a small WHILE...WEND loop, with all the real work done in routines accessed by event traps.

This can be an efficient way to structure a program, even when no real work is being done in the main program loop. Because a routine entered via an *ON event GOSUB* statement can *only* be entered when the event takes place, the structure of the program can be greatly simplified. Instead of a list of questions (Has a menu item been selected? Has the mouse button been pressed?), the main program can be shortened to an easily understandable loop, and each subroutine will have its own clearly defined function. This method is popular with professional programmers.

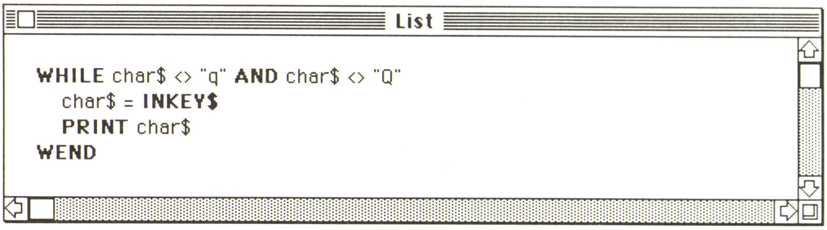
THE INKEY$ FUNCTION

We can use MOUSE(0) to give us information about the mouse. Is there a similar way to find out whether something has been typed on the keyboard? Of course, we've already learned about INPUT in Chapter 5, "Variables and Input." But whereas we are able to check the mouse with MOUSE(0) and then go off and do something else, INPUT requires that our program come to a dead stop and wait until the user has typed something in and pressed the Return key. It would be nice if there were a function that would periodically check to see if something has been typed at the keyboard, and then let our program go on its merry way, whether something had been typed or not.

The Microsoft BASIC function we're looking for is INKEY$. Like MOUSE(0), it returns a particular value. And again like MOUSE(0), no matter what the value is, program control goes on to the next instruction in line; the program is not hung up waiting for the user to press Return. However, INKEY$ is different from MOUSE(0) in that it returns a string value, rather than a number, as MOUSE(0) does. This string value is always a character, either something that was typed or, if nothing was typed, then the null string "" (the "nothing" between two sets of quotes).

FIGURE 11-11

An INKEY$ example: Characters typed scroll off the screen



```
WHILE char$ <> "q" AND char$ <> "Q"
  char$ = INKEY$
  PRINT char$
WEND
```

To see what INKEY$ does, let's look at just about the simplest program that can use this function. The listing is shown in Figure 11-11.

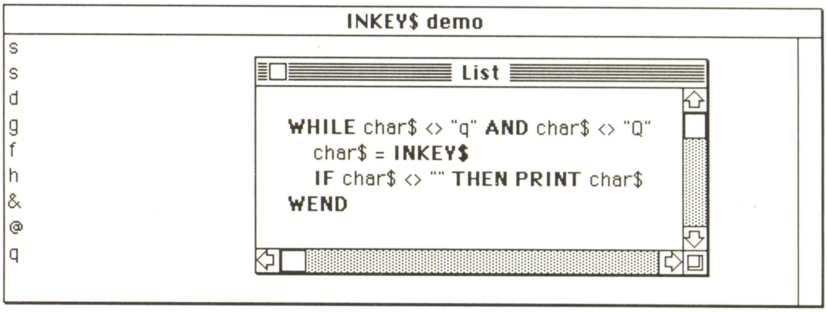
No output is shown for this program. If you type it in, run it, and type a few characters on the screen, you'll see why. Since INKEY$ doesn't wait for a Return to exit to the next program line, and since the PRINT statement thinks of the null string (which is the value *char$* has when nothing is typed) as an opportunity to print a blank line, anything you type disappears very quickly off the top of the screen as the succession of PRINTs causes the screen to scroll upward.

What we need to really see INKEY$ in action is a routine that will cause a PRINT only when there actually is a character to print. This is the purpose of the routine shown in Figure 11-12.

This routine only prints a character if the character is not the null string. As a consequence, we can now see what we've typed. This reveals a fundamental fact about INKEY$: It doesn't automatically print the character we type on the screen, the way INPUT does. That's why we need the PRINT statement with INKEY$: It's the only way to see what we typed.

FIGURE 11-12

A more complex INKEY$ example



```
s
s
d
g
f
h
&
@
q
```

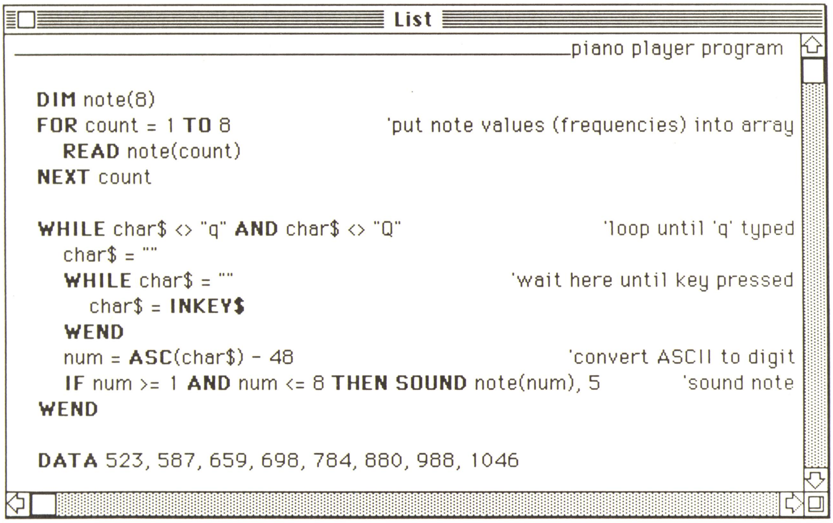
```
WHILE char$ <> "q" AND char$ <> "Q"
  char$ = INKEY$
  IF char$ <> "" THEN PRINT char$
WEND
```

Using INKEY$ to generate sound

INKEY$ is useful when we don't want to wait until the user finishes typing before doing something, and when we don't want to print out the character that was typed in. The program listed in Figure 11-13 makes use of both these characteristics.

Type in this program and run it. Since it generates no visible output, you won't get a feel for what it does if you don't try it out. When the program is running, the keyboard number keys from 1 to 8 each generate a note in the musical scale, from C up to the C an octave above. Try typing the notes. Each one sounds a tone briefly. By holding down a key, you'll get a trilling effect produced by a succession of short notes.

The program first puts a series of numbers representing the frequencies of the notes into the array *note,* using a READ statement. This makes it possible to access a particular value very easily. The outer WHILE...WEND loop makes sure the INKEY$ statement is repeated until the piano player gets tired. The inner WHILE...WEND loop waits for a key to be pressed. As long as the character returned by INKEY$ is the null string, this loop keeps cycling. As soon as a character is typed, control falls through to the next two statements. The first of these statements converts the ASCII value of the character to an integer value from 1 to 8 by subtracting 48, which is the ASCII value of the character *0,* as we learned in the last chapter. Then, in the IF...THEN statement, the program checks to make sure the number was in the range 1 to 8. If so, it sounds the appropriate tone by taking the pitch value from the array. If not, it cycles again, looking for another character.



```
                                                  piano player program

DIM note(8)
FOR count = 1 TO 8                   'put note values (frequencies) into array
  READ note(count)
NEXT count

WHILE char$ <> "q" AND char$ <> "Q"                          'loop until 'q' typed
  char$ = ""
  WHILE char$ = ""                                    'wait here until key pressed
    char$ = INKEY$
  WEND
  num = ASC(char$) - 48                                     'convert ASCII to digit
  IF num >= 1 AND num <= 8 THEN SOUND note(num), 5                    'sound note
WEND

DATA 523, 587, 659, 698, 784, 880, 988, 1046
```

FIGURE 11-13

The piano-player program

REFRESH-YOURSELF WINDOW

In this chapter we've learned several techniques for introducing the mouse into our programs. We've also learned about a new way to structure our programs, using event trapping. Finally, we've learned about a powerful new way to get characters from the keyboard: INKEY$. These techniques should enhance your programming skill, making your programs more professional and exciting for the user.

Menus

In the preceding chapters, we explored many different BASIC statements and functions, and we learned a lot about writing programs for the Macintosh. We also learned about some features unique to the Mac, like the mouse and high-resolution graphics. But these are only a couple of the wonderful features of the Mac that are available to Microsoft BASIC.

In the next few chapters, we'll learn even more about how to control the special features of the Macintosh with BASIC. We'll show you how to use the Macintosh *user interface* to make your progams "act like" commercial Mac programs. We'll learn how to create and use menus, buttons, and windows in our BASIC programs. After all, these unique features are what make the Macintosh so popular, so our programs should take advantage of them.

Microsoft BASIC for the Macintosh includes several special statements and functions that provide the programmer access to the menu aspect of the user interface. We're going to cover these in this chapter. These statements and functions are different from anything we've seen before, and require that our programs follow a certain "architecture" that may look strange to you. But don't worry—we'll take it slowly and explain things as we go along.

WHAT IS A MENU?

A menu is a "user friendly" way of letting you communicate with the Macintosh. We've been using menus all along in our discovery of Microsoft BASIC. Every time we run a program, we do so by choosing Start from the Run menu. To save a program on the disk, we choose Save or Save As... from the File menu. It is this concept of *choice* that makes the menu such a powerful programming tool. We don't have to remember fancy commands made up of combinations of file names and tricky punctuation like:

```
SAVE "BIN BASIC:Graphics", P
```

Instead, we are presented with a list of choices we can make, which eliminates any possibility of typing mistakes, ambiguity, or error.

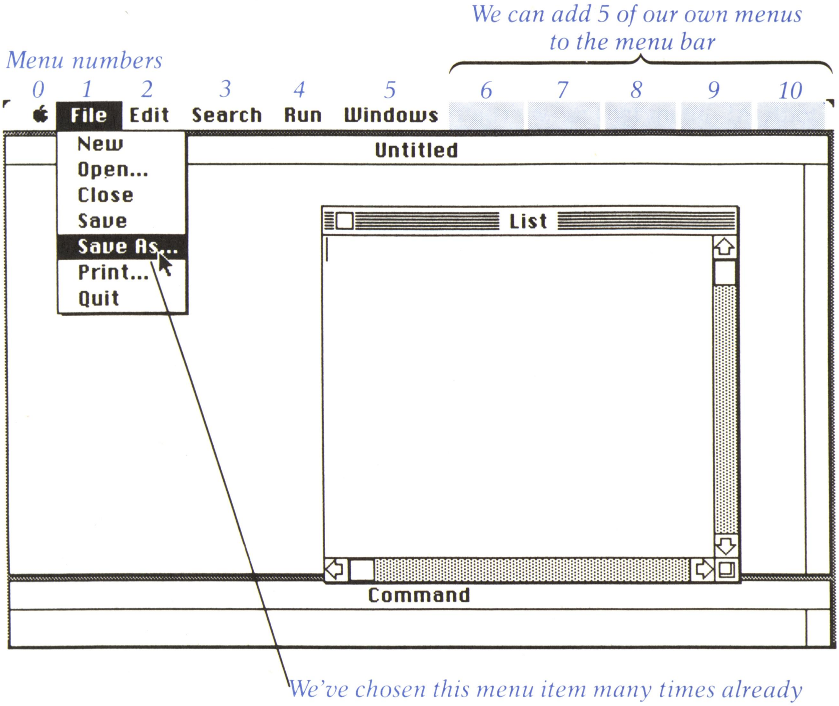
Now let's briefly review what we already know about menus and how to use them.

THE BASIC MENU BAR

The menus we've been using in BASIC are like those pull-down maps and charts that hang from the top of the blackboard in every classroom. But on the Macintosh, the menus "hang" from the *menu bar* across the top of the screen. We've already worked with most of the six menus provided on the BASIC desktop. However, as you can see in Figure 12-1, there is room on the menu bar for five additional menus, numbered six through ten. You can

FIGURE 12-1

The BASIC menu bar with its six menus


Menu numbers
0 1 2 3 4 5 6 7 8 9 10
File
Edit
Search
Run
Windows
New
Open...
Close
Save
Save As...
Print...
Quit
Untitled
List
Command

We can add 5 of our own menus to the menu bar

We've chosen this menu item many times already

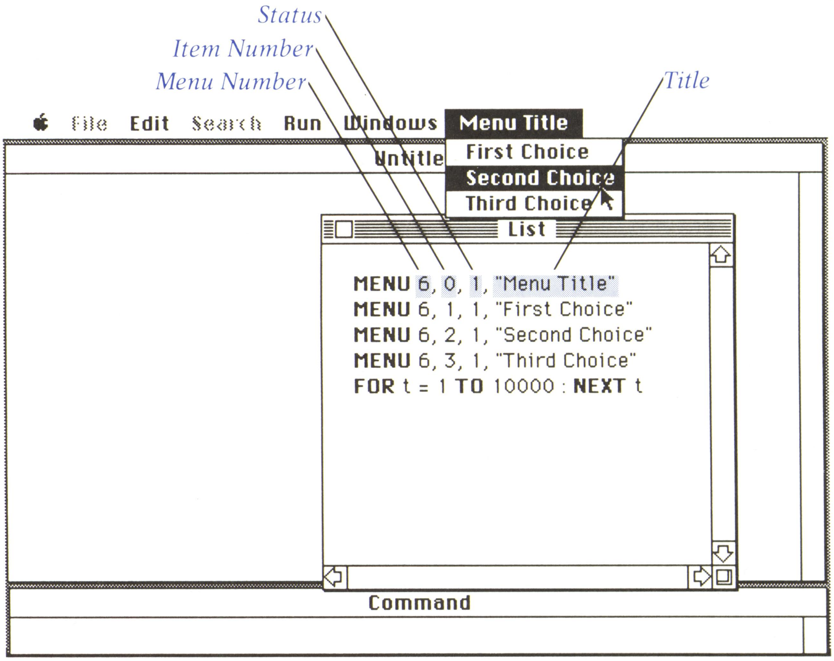
create menus in these unused positions, or even in the positions used by BASIC (except for the 0th, or Apple menu). Let's see how easy it is to create our own menus in a BASIC program.

CREATING A MENU: THE MENU STATEMENT

Menus are created with the MENU statement, although it takes several MENU statements to create an entire menu. In fact, we need one statement to define the menu itself, and one additional statement for each item on the menu. A simple example of the MENU statements in action is shown in Figure 12-2 on the following page.

As you can see in the figure, a menu containing three choices is created with four MENU statements. (The FOR…NEXT statement is simply a time-delay loop; without it, the menu bar would be reset at the end of the program.) Let's take a closer look at the MENU statement itself. Its general syntax is:

MENU *Menu Number, Item Number, Status, Title$*



```
MENU 6, 0, 1, "Menu Title"
MENU 6, 1, 1, "First Choice"
MENU 6, 2, 1, "Second Choice"
MENU 6, 3, 1, "Third Choice"
FOR t = 1 TO 10000 : NEXT t
```

FIGURE 12-2

Creating a simple menu

Before we get into more actual examples of the MENU statement, let's take a moment to discuss the meaning of the four parameters we need to supply in order to produce our menus.

Menu Number: The menu ID number

Each menu is assigned an ID number, which is the same as its *position* in the menu bar. The parameter *Menu Number* contains this ID number, which BASIC uses to distinguish between the individual menus. Remember, the standard Microsoft BASIC menu bar already contains menus 0 through 5. While it's possible to assign our menu an ID number in that range, the result is the temporary elimination of the corresponding BASIC menu. Since you're just learning BASIC, we suggest that you begin numbering your menus with ID number 6, so that your program will leave the standard BASIC menu intact. Later, we'll show you how to create an entirely custom-made menu bar.

Item Number: The item ID number

Each choice that can be made from a menu is called a *menu item*, and requires one MENU statement in the program to define it. The *Item Number* parameter specifies *which* menu item is being defined by the MENU statement. We must have one MENU statement just to define the menu title itself, and in that statement the *Item Number*'s value must be 0. This places the menu title on the menu bar. Next, we must have a MENU statement to define each item that is to appear in that menu. Each item's *Item Number* value corresponds to the position of the item within that menu. In other words, the first menu item should have *Item Number* equal to 1, the second equal to 2, and so on. It's possible to have a menu with up to 20 items on it, but it's unlikely that any of our programs will require this abundance.

Status: Controlling the item's appearance

You may have noticed that the menu items we've been choosing throughout the book (Open, Close, Save, and so forth) seem to change their appearance in different situations. Sometimes menu items are "dimmed" to show that they are not available at that time. If you've used MacWrite or MacPaint, then you've probably noticed that some menus have a check mark preceding the currently selected item. These menu attributes are controlled by the *Status* parameter. There are three different states that a menu item can be in:

0—Dimmed (item cannot be chosen)
1—Normal (item can be chosen)
2—Check mark preceding selected item

The "normal" status of a menu item is 1. This means that it appears normally (black letters on a white background) and can be chosen from the menu. Later in this chapter we'll see how to change the status of a menu item to affect the operation of our programs.

Title$: Naming the menu and items

The *Title$* parameter contains the actual title description for either the menu's title or the individual items on the menu, and consequently must be a string constant, variable, or expression. When the *Item Number* parameter is 0 (the menu title), the title in *Title$* is displayed in the menu bar at the top of the screen. Each MENU statement that defines a menu item *must* contain the description of the menu item in its *Title$* parameter.

Now that we've explored the general syntax of the MENU statement, let's learn how we can put our menus to work for us in our programs once we've created them.

PROGRAMMING MENUS

There are two things our programs must know in order to use a menu we've created:

- The *menu* from which the item was chosen.
- The *item* that was chosen.

To obtain these two pieces of information, our programs must use a *query* feature of the Macintosh user interface: the MENU *function*. The MENU *function* must not be confused with the MENU *statement*. The way to remember the difference is that the *function returns a value*, which the program will use, while the *statement creates the menu*.

The MENU function works a lot like the MOUSE function we've used in previous chapters. But whereas the MOUSE function reports on the status and position of the mouse, the MENU function reports the status of the menus by returning a number (which we'll explain in just a moment). Now let's see what the MENU function looks like.

Menu selection: "Polling" the MENU function

The MENU function is easy to use. It requires only one argument, which tells it what information we are looking for. The general form of the MENU function is:

Answer = **MENU**(*Question*)

There are two *questions* we can ask the MENU function, and they are numbered as follows:

0—Which *menu* was the item chosen from?
1—Which *item* was chosen from that menu?

Let's take these questions in order, and see how we can use the MENU function to make our programs smarter.

MENU(0): Which menu?

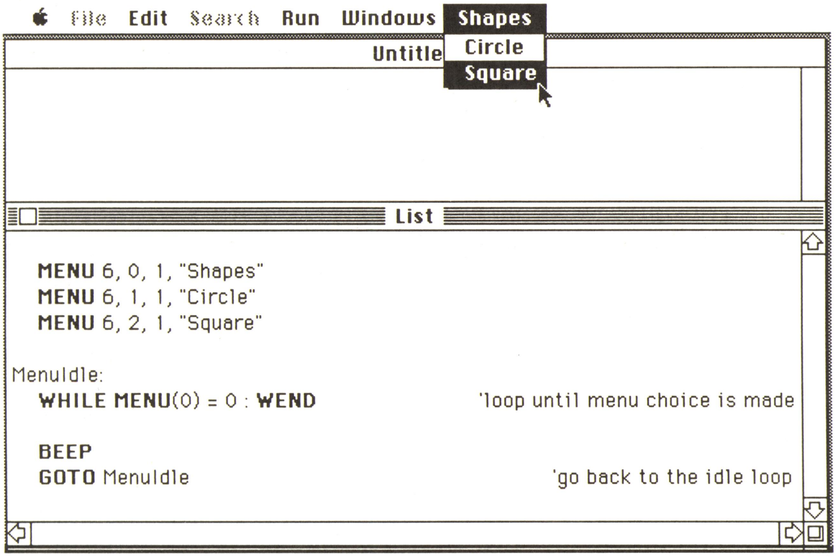
The first thing our program needs to know is whether or not the user has chosen something from one of the menus. This is accomplished using the MENU function with an argument of 0. When we use MENU(0) in a statement, it checks to see if the user has made a choice from a menu and returns a value accordingly:

(0)—No menu choice has been made
(1-10)—The menu ID number of the menu from which a choice was made

The MENU function will return a value of 0 if no menu choice has been made, or a value between 1 and 10, indicating which menu was chosen from. Let's see how we can use this concept in our programs.

The three MENU statements and the other lines shown in Figure 12-3 begin the example program we'll be using throughout this chapter to illustrate the typical use of menus. The program itself will simply perform some sound, graphic, and text output to show the effects of menu control.

As in our first example in Figure 12-2, the menu we've created is assigned menu number 6 to avoid conflicting with BASIC's standard menu bar. This menu will control the graphic display of two shapes: a circle and a square. The first MENU statement defines the menu title as *Shapes,* since the *Item Number* parameter is 0. The following two statements define the two choices that will be available on the menu: Circle and Square. The *Status* parameter is the same for all three MENU statements. The value 1 specifies that all the items can be *chosen.* That's why they appear normal, not dimmed or followed by check marks.



```
  MENU 6, 0, 1, "Shapes"
  MENU 6, 1, 1, "Circle"
  MENU 6, 2, 1, "Square"

MenuIdle:
  WHILE MENU(0) = 0 : WEND                  'loop until menu choice is made

  BEEP
  GOTO MenuIdle                                    'go back to the idle loop
```

FIGURE 12-3

Polling the MENU function

Next comes the *Menu Idle* subroutine. Since we know that the MENU function will return a value of 0 if no menu choice has been made, we'll use the WHILE...WEND statement to produce an "idle loop" that just keeps looping as long as the MENU function returns 0. This is like putting our program on a treadmill until a menu choice is made. This technique is much like the one we used in earlier chapters with the MOUSE function, and can be accomplished with one line of BASIC:

```
WHILE MENU(0) = 0 : WEND
```

The moment a choice is made from the menu, the MENU function returns the number of the menu from which the choice was made. This causes the WHILE...WEND loop to end, since the value is no longer 0. BASIC then executes the next statement, in this case a BEEP and then a GOTO that sends the program back to the idle loop.

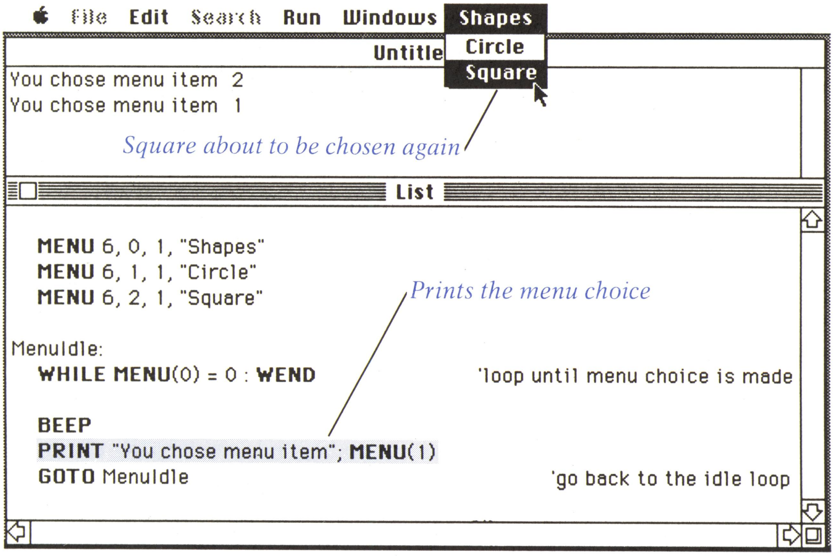
(In general, we do not encourage the use of the GOTO statement in larger programs. However, for the sake of clarity here, and to focus our attention on the MENU statement and function, we have simplified the rest of the program by using GOTO.)

Well, that's a start! At least we can tell *when* the user makes a choice from a menu. But we didn't really find out *what* choice was made. Let's see how we can do that.

MENU(1): Which item?

Once we've used the MENU(0) function to determine that a menu choice was made and which menu the choice was made from, we can use the MENU(1) function to determine which *item* was chosen from the menu. It's important to remember that we *must* use the MENU(0) function before MENU(1) will return a valid choice. Let's improve our example program to do something useful.

As shown in Figure 12-4, we've added a PRINT statement to our program. Now when a menu choice is made and the program falls out of the WHILE...WEND loop, it displays the value returned by the MENU(1) function. This tells us which item was chosen from the menu.


File
Edit
Search
Run
Windows
Shapes
Circle
Square
Untitled
List

```
You chose menu item  2
You chose menu item  1
```

Square about to be chosen again

```
   MENU 6, 0, 1, "Shapes"
   MENU 6, 1, 1, "Circle"
   MENU 6, 2, 1, "Square"

MenuIdle:
   WHILE MENU(0) = 0 : WEND                'loop until menu choice is made

   BEEP
   PRINT "You chose menu item"; MENU(1)
   GOTO MenuIdle                             'go back to the idle loop
```

Prints the menu choice

FIGURE 12-4

Printing the menu choice using MENU(1)

ARCHITECTURE OF A MENU-DRIVEN PROGRAM

If you've started to notice some similarities in our example programs, it's because using menus requires a particular program *architecture.* Figure 12-5 illustrates this program architecture.

There are three main sections to any program that is going to use menus. They are:

1. Initialize menus.
2. Wait loop until menu choice.
3. Process menu choice.

We can see these three sections in our example program in Figure 12-4, and we'll see this architecture maintained even as our programs get larger and more complex.

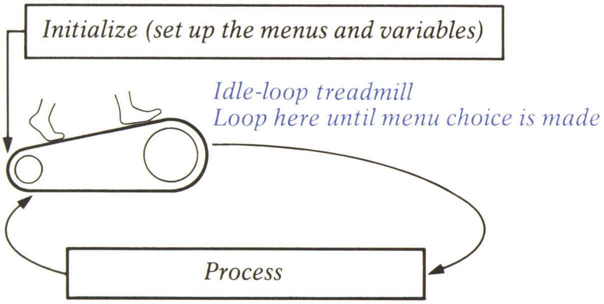

Initialize (set up the menus and variables)
Idle-loop treadmill
Loop here until menu choice is made
Process

FIGURE 12-5

Architecture of a menu-polling program

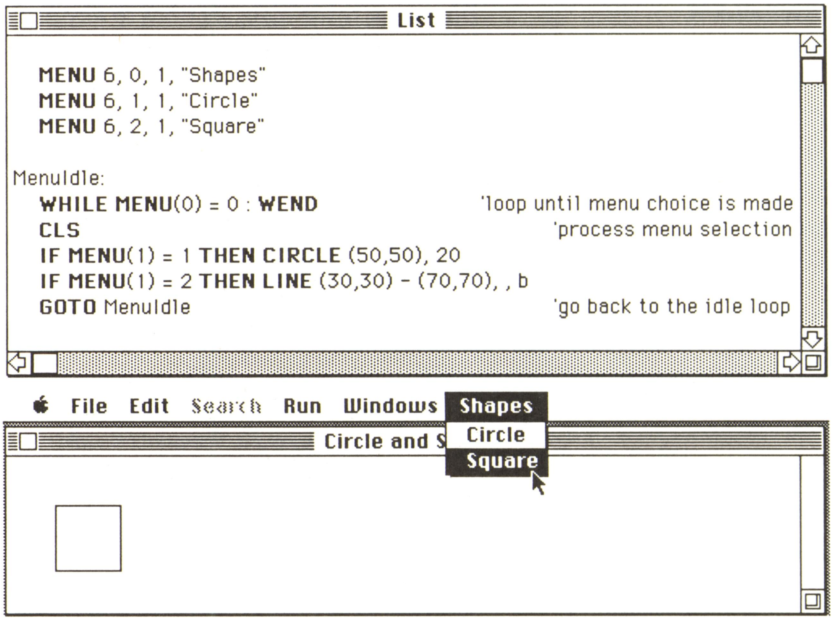
Now we can tell when a menu choice is made, from which menu, and which item was chosen. Let's see how we can use this information in our programs.

WHAT ACTION TO TAKE?

There are many possible reasons for choosing from a menu, but it's usually because the user wants the program to do something. Let's expand our example program now to display some graphics on the screen according to the menu choice.

Simple menu choices with the IF...THEN statement

When the action to be taken in response to a menu choice is fairly simple, the easiest way to make it happen is to use the decision-making IF...THEN statement. For example, in Figure 12-6, we have replaced our original PRINT statement with two IF...THEN statements. Each one checks the MENU(1) function to see if the item number it returns is equal to either 1 or 2. If item 1 was chosen, a circle is drawn. If item 2 was chosen, the LINE statement is used to draw a square. After this dramatic graphics display, the GOTO statement directs the program back to the idle loop where it waits for another menu choice to be made.



```
   MENU 6, 0, 1, "Shapes"
   MENU 6, 1, 1, "Circle"
   MENU 6, 2, 1, "Square"

MenuIdle:
   WHILE MENU(0) = 0 : WEND                         'loop until menu choice is made
   CLS                                                    'process menu selection
   IF MENU(1) = 1 THEN CIRCLE (50,50), 20
   IF MENU(1) = 2 THEN LINE (30,30) - (70,70), , b
   GOTO MenuIdle                                          'go back to the idle loop
```

FIGURE 12-6

A simple menu choice

Controlling the menu bar

This program is getting smarter all the time. With just a few lines of BASIC we've created a menu that we can use to direct the operation of our program. But if you try running this program, you'll notice something odd about the way our new menu works. The menu title remains highlighted, indicating that the menu is in use. Let's see why this happens and how we can control it with our programs.

The reason that it's so easy for us to use menus in our programs is that BASIC and the Macintosh do a lot of the work for us. All of the Macintosh menus operate in the same way: We use the mouse to "pull down" the menu, drag to the item of our choice, then release the mouse button to make our choice and cause the menu to roll back up into the menu bar. The highlighting we have noticed is a "fringe benefit" of the way menus work on the Macintosh. When the user makes a menu choice, the title is highlighted on the menu bar to provide visual feedback to the user. You can see the effect of this in Figure 12-7.

Nothing comes without its price. Since BASIC and the Macintosh give us this marvelous feature of highlighting a menu title, we have to "unhighlight" it when our program has completed the requested operation. This is accomplished by an additional MENU *statement* following our two IF...THEN statements. See Figure 12-8.

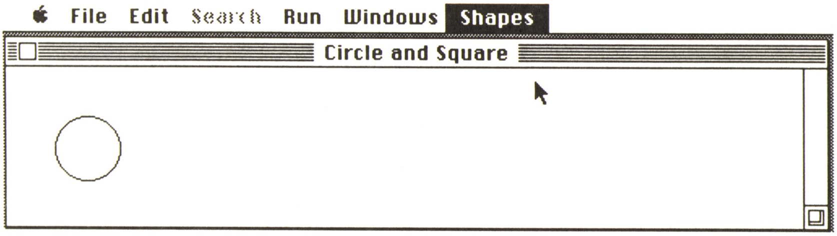
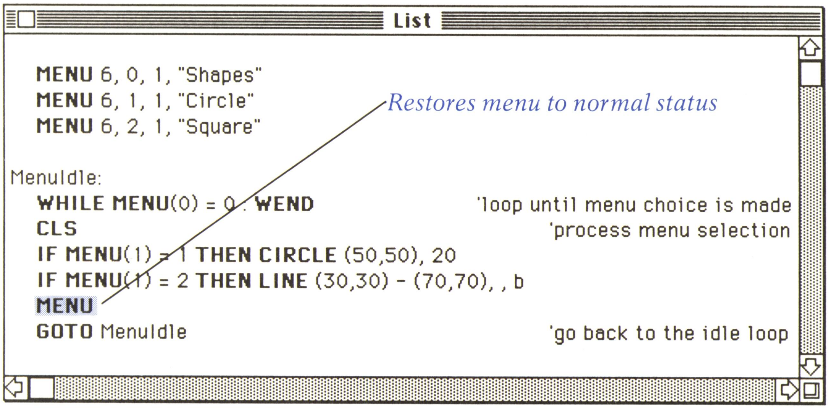


FIGURE 12-7

The menu title remains selected after a choice is made



```
    MENU 6, 0, 1, "Shapes"
    MENU 6, 1, 1, "Circle"
    MENU 6, 2, 1, "Square"

Menuldle:
    WHILE MENU(0) = 0 : WEND                    'loop until menu choice is made
    CLS                                                'process menu selection
    IF MENU(1) = 1 THEN CIRCLE (50,50), 20
    IF MENU(1) = 2 THEN LINE (30,30) - (70,70), , b
    MENU
    GOTO Menuldle                                      'go back to the idle loop
```

Restores menu to normal status

FIGURE 12-8

Unhighlighting the menu title

Notice that this new MENU statement stands alone without specifying any of the parameters we've used before. The reason for this is that the MENU statement by itself returns the currently selected menu to a normal state. We'll use this method in all our programs that use menus.

Well, now we've managed to get our program to produce shapes according to our menu choice. It was easy, using the IF...THEN statements to check the MENU(1) function to find out which item was chosen. But what if we wanted to do more than just draw some shapes? There are better tools than the IF...THEN statement; let's see what they are.

Multiple choice: The ON...GOTO and ON...GOSUB statements

While IF...THEN statements are powerful programming tools in BASIC, they have some drawbacks. For example, if you want to have your program perform several different operations as the result of a menu choice, you will have to write long IF...THEN statements like this:

IF MENU(1) = 1 **THEN** [*Statement* : *Statement* : *Statement* : ...]

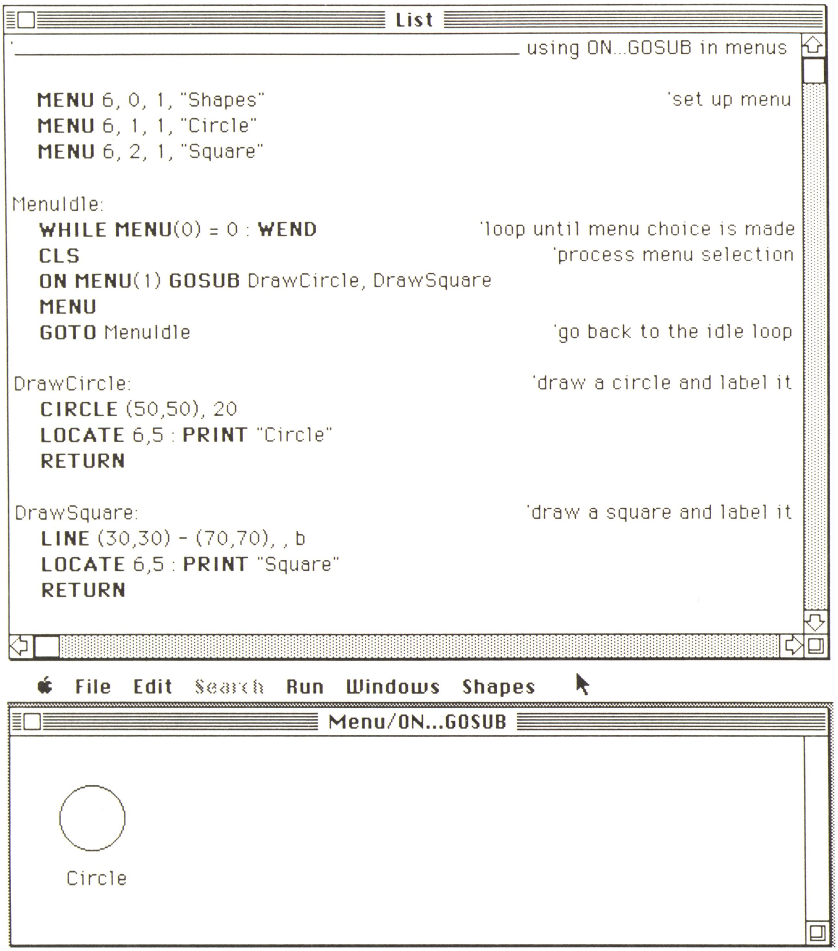
This is not a good programming technique since it makes a program difficult to understand. A much better way of dealing with this is to use the ON...GOSUB statement to direct the program to a subroutine for each menu choice. We can put any BASIC statements we want in the subroutines, and they will be executed whenever the user makes the appropriate choice from the menu. Let's see how this works.

Our example program, shown in Figure 12-9, is growing by leaps and bounds! We've removed the two IF...THEN statements because we want to do more with our program. Besides displaying the circle or square, our program will also label them on the screen using PRINT statements, as shown in the Output window of the figure.

The ON...GOSUB statement uses the function MENU(1) as its index. When MENU(1) returns a value of 1 (meaning that Circle was chosen), the program branches to the *DrawCircle* subroutine. When the function returns a value of 2 (meaning that Square was chosen), the program branches to the *DrawSquare* subroutine.

When the program returns from either of the two subroutines, another MENU statement is used to unhighlight the menu bar. This is a great way to use menus! We can put any BASIC statements we want inside the two subroutines, and they will be executed whenever we make the appropriate choice from the menu. Our programs can do simple operations using the IF...THEN statement, but will probably end up using the ON... GOSUB method most of the time. That's because it makes the program more understandable.

Now we've learned how to use a menu in our programs to make them act like "real" Macintosh programs. The user can direct the operation of the program with the mouse, rather than by commands typed at the keyboard. This makes our programs more reliable, since much of the ambiguity of user choice is removed.



```
'__________________________________________ using ON...GOSUB in menus

   MENU 6, 0, 1, "Shapes"                                 'set up menu
   MENU 6, 1, 1, "Circle"
   MENU 6, 2, 1, "Square"

MenuIdle:
   WHILE MENU(0) = 0 : WEND               'loop until menu choice is made
   CLS                                        'process menu selection
   ON MENU(1) GOSUB DrawCircle, DrawSquare
   MENU
   GOTO MenuIdle                             'go back to the idle loop

DrawCircle:                                  'draw a circle and label it
   CIRCLE (50,50), 20
   LOCATE 6,5 : PRINT "Circle"
   RETURN

DrawSquare:                                  'draw a square and label it
   LINE (30,30) - (70,70), , b
   LOCATE 6,5 : PRINT "Square"
   RETURN
```

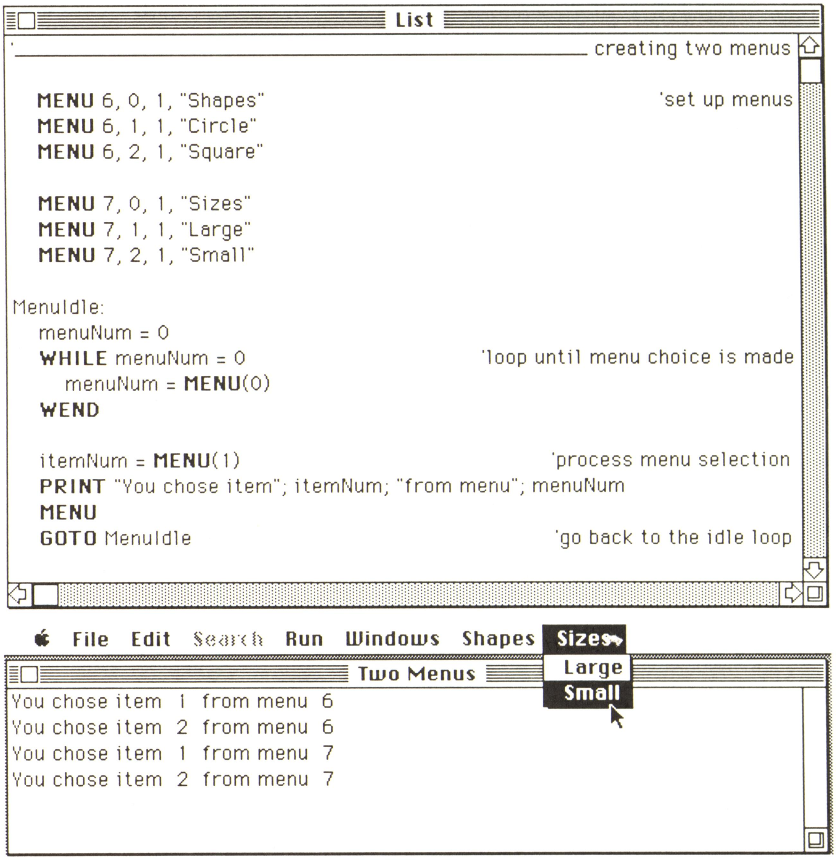
FIGURE 12-9

A program using ON...GOSUB to perform multiple actions with a menu choice

USING MORE THAN ONE MENU

All of our examples so far have dealt with only one menu. But remember: There are five extra menu slots available to our BASIC programs, and each one will operate as a completely independent Macintosh menu. Let's see how we might use more than one menu in a program.

As we've seen while using BASIC, it's often necessary to have more than one menu controlling the operations of the program. Usually, each menu has some particular application area (such as editing or searching), and the items on that menu are related in some way. As shown in Figure 12-10, creating multiple menus is easy: We just use more MENU statements. However, interpreting the user's menu choices must be approached differently.



```
'_____________________________________________________ creating two menus

   MENU 6, 0, 1, "Shapes"                                    'set up menus
   MENU 6, 1, 1, "Circle"
   MENU 6, 2, 1, "Square"

   MENU 7, 0, 1, "Sizes"
   MENU 7, 1, 1, "Large"
   MENU 7, 2, 1, "Small"

MenuIdle:
   menuNum = 0
   WHILE menuNum = 0                      'loop until menu choice is made
      menuNum = MENU(0)
   WEND

   itemNum = MENU(1)                                'process menu selection
   PRINT "You chose item"; itemNum; "from menu"; menuNum
   MENU
   GOTO MenuIdle                                   'go back to the idle loop
```

```
You chose item  1  from menu  6
You chose item  2  from menu  6
You chose item  1  from menu  7
You chose item  2  from menu  7
```

FIGURE 12-10

A program that creates two menus

Which menu?

Programs that use only one menu need only know when the user has made a menu choice. As we've seen, this is easily accomplished by using the MENU(0) function in a WHILE...WEND loop. But when our programs use more than one menu, we must find out *which* menu the user chose from.

Remember, the MENU(0) function returns the ID number of the menu from which the choice was made, or returns 0 if no menu choice was made since the last time the program used MENU(0). If our program needs to know which menu was chosen from, then we must assign the value returned by MENU(0) to a variable, in this case *menuNum,* which is then used to make further references to the menu choice.

In a similar manner, we must use the MENU(1) function to find out which menu item was chosen. Here again, since we may need to use this information while we wait for another menu choice, we'll assign the value returned by MENU(1) to a variable, *itemNum.* Then, we can use both the menu number and the item number. The Output window shown in Figure 12-10 illustrates how the program displays which item was chosen from which menu by printing the values in the two variables. Of course, in a real program the values would be used to direct BASIC's processing in more useful ways, such as branching to other subroutines.

Now that we know how easy it is to use two menus, let's incorporate this knowledge into our Shapes program.

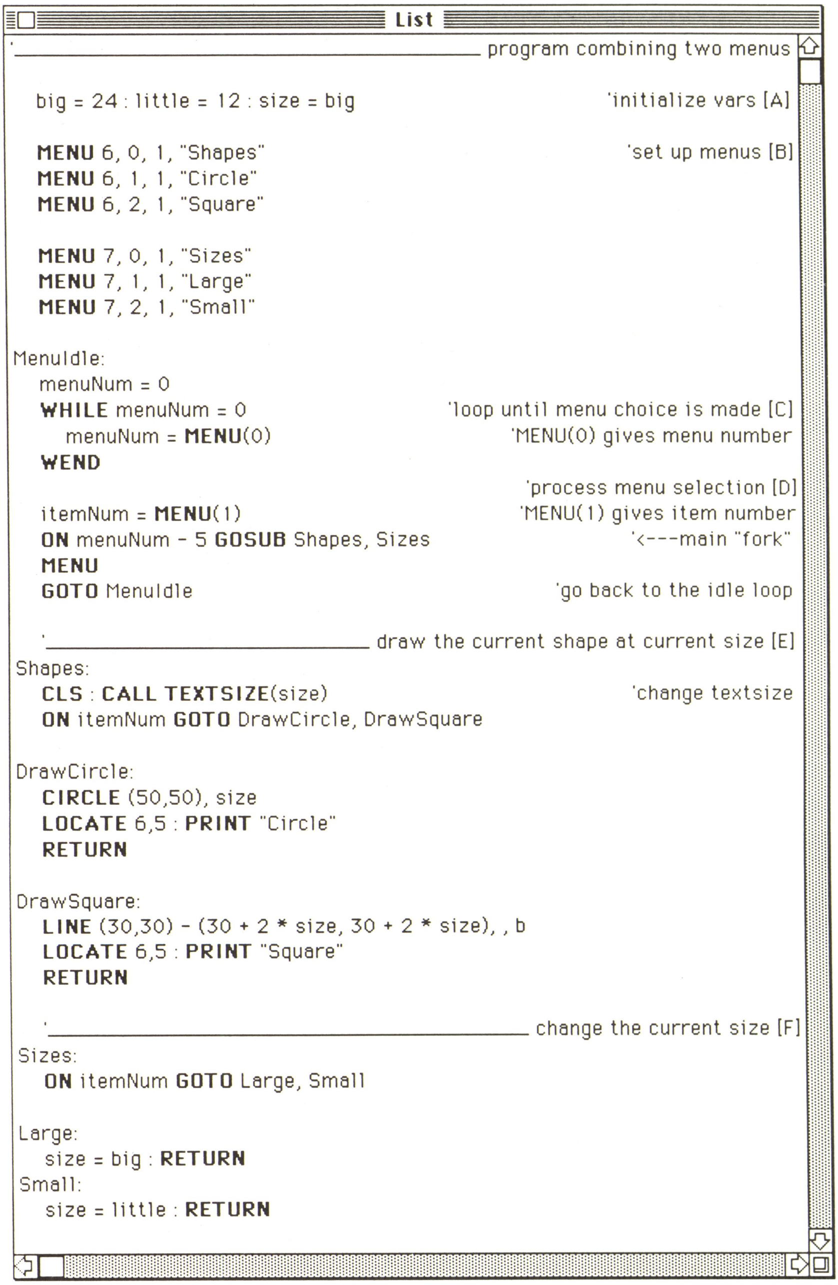
Combining two menus

In the next example we are going to expand our earlier shape-drawing program to let us specify whether we want a *large* or *small* shape. We'll use a second menu to allow this choice. Then we'll use a little bit of everything we've learned so far to make the program display either large or small text and graphics.

Before we look at this next program, prepare yourself for a bit of a surprise. Our program is about to increase in overall size and apparent complexity. However, you will notice that it still follows the same architecture we mentioned earlier, having three major sections: *initialize menus, wait until menu choice,* and *process menu choice.* While the first two sections reflect the addition of new MENU statements, it is the increased processing objective (shapes of selectable size) that is responsible for the growth of our program.

We'll digress briefly here to introduce one new statement before plunging into the heart of the program: the ON...GOTO statement. As you may have guessed, this statement is similar to the ON...GOSUB statement, except that instead of executing a GOSUB statement to branch to a label (or line number) calculated from the numerical expression in the statement, ON…GOTO simply jumps to the specified label using a GOTO.

To help you follow along with the program, we've marked each section at the right margin of the program listing shown in Figure 12-11



```
'_________________________________________ program combining two menus

   big = 24 : little = 12 : size = big                 'initialize vars [A]

   MENU 6, 0, 1, "Shapes"                                 'set up menus [B]
   MENU 6, 1, 1, "Circle"
   MENU 6, 2, 1, "Square"

   MENU 7, 0, 1, "Sizes"
   MENU 7, 1, 1, "Large"
   MENU 7, 2, 1, "Small"

MenuIdle:
   menuNum = 0
   WHILE menuNum = 0                      'loop until menu choice is made [C]
      menuNum = MENU(0)                           'MENU(0) gives menu number
   WEND
                                                 'process menu selection [D]
   itemNum = MENU(1)                              'MENU(1) gives item number
   ON menuNum - 5 GOSUB Shapes, Sizes                       '<---main "fork"
   MENU
   GOTO MenuIdle                                    'go back to the idle loop

   '______________________________ draw the current shape at current size [E]
Shapes:
   CLS : CALL TEXTSIZE(size)                                 'change textsize
   ON itemNum GOTO DrawCircle, DrawSquare

DrawCircle:
   CIRCLE (50,50), size
   LOCATE 6,5 : PRINT "Circle"
   RETURN

DrawSquare:
   LINE (30,30) - (30 + 2 * size, 30 + 2 * size), , b
   LOCATE 6,5 : PRINT "Square"
   RETURN

   '_______________________________________________ change the current size [F]
Sizes:
   ON itemNum GOTO Large, Small

Large:
   size = big : RETURN
Small:
   size = little : RETURN
```

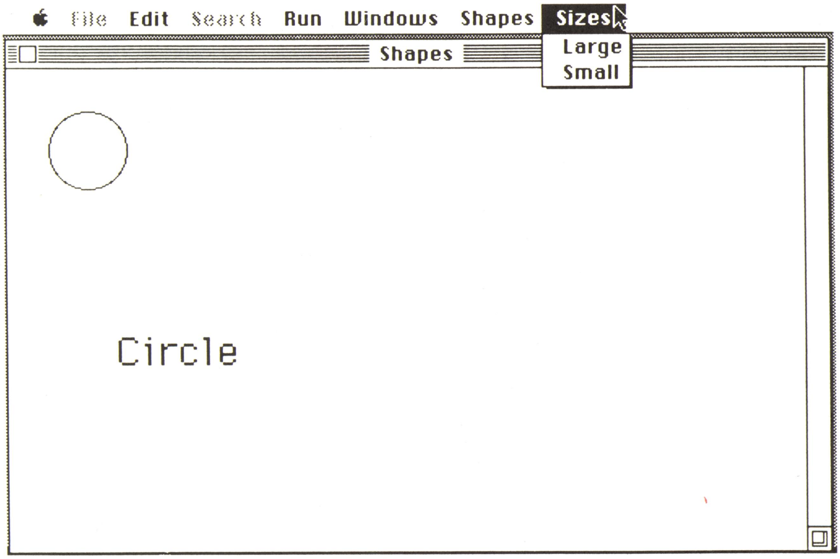
FIGURE 12-11

A program combining two menus

(continued)

FIGURE 12-11

(continued)



with a capital letter enclosed in square brackets, which is referenced in the following sections.

Section A. To get our program to draw either large or small shapes, we'll set up some variables, *big, little,* and *size,* which will be used later to control the size of the text and graphics. You'll notice that *size* is initially set to *big.* This is called the "default" size, and will be used by the program if the user does not choose a size from the menu.

Section B. Two sets of MENU statements are used to create the two menus generated by this program. It's nice to separate the sets the way we've done here, to make their purpose more clear to anyone looking at our program. The first set creates the Shapes menu and the second creates the Sizes menu.

Section C. The *MenuIdle* loop is the more complex version we just learned about, which saves the ID number of the last menu chosen from in the variable *menuNum.*

Section D. When a menu choice is made, the program "falls" into this section. The ID number of the menu from which a choice was made is already saved in the variable *menuNum.* Here, the MENU(1) function gets the item number of the menu choice and saves it in the variable *itemNum.* The next statement:

```
ON menuNum - 5 GOSUB Shapes, Sizes
```

forms the main "fork" in our program. It is here that our program will branch to either draw shapes, or change the default size.

Notice that the index for this ON...GOSUB statement is an expression that subtracts the constant 5 from the value stored in the variable *menuNum.* Remember, we assigned ID numbers 6 and 7 to our two new menus. The ON...GOSUB statement wants an index value of 1 or greater. So that's why we use an expression to subtract 5 from whatever menu ID number was returned by the MENU(0) function. When a choice is made from menu number 6, for example, the index of the ON...GOSUB statement will be equal to 1, and so on.

You might be wondering what would happen if an item from one of the five BASIC menus were chosen while the program was running: Would MENU(0) return the number 1, 2, 3, 4, or 5? You might conclude that if, for example, Paste were chosen from the Edit menu, *menuNum* would be 2 and the index for the ON...GOSUB would be $2-5=-3$. We learned in an earlier chapter that an ON...GOSUB statement will cause an error if a negative number is used in its index. So would our program stop and display an error message if one of the five BASIC menus were chosen?

Well, it so happens that MENU(0) returns a useful number *only* for the menus we define in our program, because if any of BASIC's default menus are selected, MENU(0) will return a 0. Since the WHILE...WEND wait loop cycles until *menuNum* is not equal to 0, the only values that *menuNum* can have outside of the loop in this case are 6 and 7, the ID numbers of the menus we've defined earlier.

Once our program receives one of these values it branches to the appropriate subroutine. Each menu has two items on it, and our program should perform according to which item is chosen.

Section E. The *Shapes* subroutine looks similar to that used in an earlier version of this program, but we have added some statements to make the shapes different sizes. We used the variable *size* for the radius in the CIRCLE statement, and also to calculate one of the corners of the square drawn by the LINE statement.

Section F. This section contains the new statements we've added to allow the user to specify if the shape should be drawn large or small. If the user made a choice from the Sizes menu, then the program branches to the subroutine that is labeled *Sizes.* Here, as in the *Shapes* subroutine, an ON...GOTO statement directs the program to the proper statements. The variable *itemNum,* which was set earlier by the MENU(1) function, is used as the index to this ON...GOTO statement.

The action taken by the program when a choice is made from the Sizes menu is quite short. In either case, the program simply redefines the value of the variable *size* and returns to the point of the last subroutine call. This was the "main fork" of the program, and the statement following the fork is the expected MENU statement which "unhighlights" the menu-bar name before the program goes back into the idle loop.

The next time a choice is made from the Shapes menu, the program will draw in the size last chosen from the Sizes menu. If you run this program now, you'll be given the output shown in Figure 12-11. We can now draw either a large or small circle or square.

In looking at the sample output, one idea that comes to mind is that it would be helpful to know whether the default size is large or small. We'll learn how to do this, and more, later on when we discuss check marks and dimming. Let's take a break from this program for now, though, and look at some optional methods of creating menus in our programs.

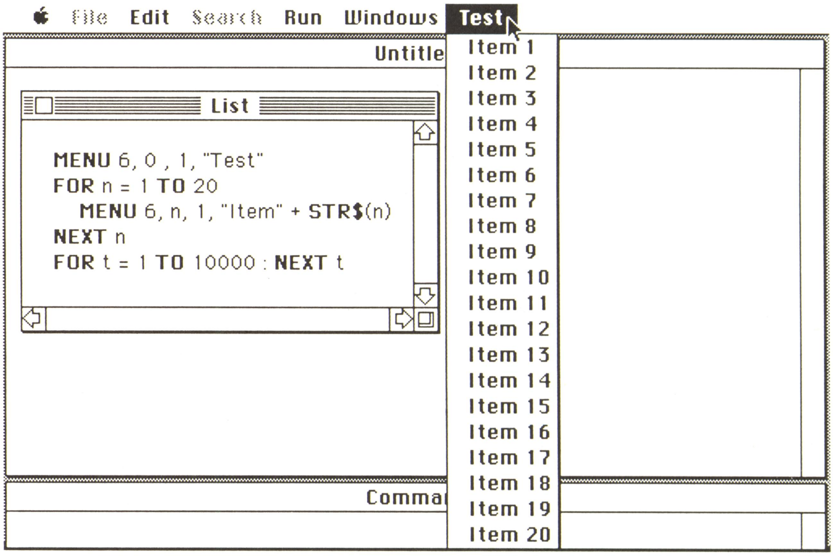
ALTERNATE METHODS FOR CREATING MENUS

The menus we've used in our example programs have been quite simple and straightforward. Since there are only two items on each menu, they were easy to create with just a few MENU statements. But what if we need a more complicated menu with many items? Happily, there are other shortcut methods for creating menus. The FOR...NEXT loop can be used to create orderly menus in a hurry. We can put our menu titles into DATA statements and then READ them when we create the menu. We can also make unusual menus if we want, or replace the standard menus of Microsoft BASIC. Let's take a look at each of these possibilities.

Fast menus with FOR...NEXT

Early in this chapter we mentioned that you can have up to 20 items on any given menu. Chances are that you won't ever need that many items on a menu, but if you do, a quick and easy way of creating the menu is shown in Figure 12-12 on the next page.

As you can see in the figure, it really is possible to make a menu with 20 items. Using a FOR...NEXT loop containing a single MENU statement, we were able to create this menu in a flash. Observe that the loop's index *n* is used as the *itemNum* parameter for the MENU statement. Also, the *Title* parameter is composed of a string expression concatenating the string-constant *Item* with the string equivalent of the loop index *n*.


File
Edit
Search
Run
Windows
Test
Untitle
List
Comma
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10
Item 11
Item 12
Item 13
Item 14
Item 15
Item 16
Item 17
Item 18
Item 19
Item 20

```
MENU 6, 0 , 1, "Test"
FOR n = 1 TO 20
   MENU 6, n, 1, "Item" + STR$(n)
NEXT n
FOR t = 1 TO 10000 : NEXT t
```

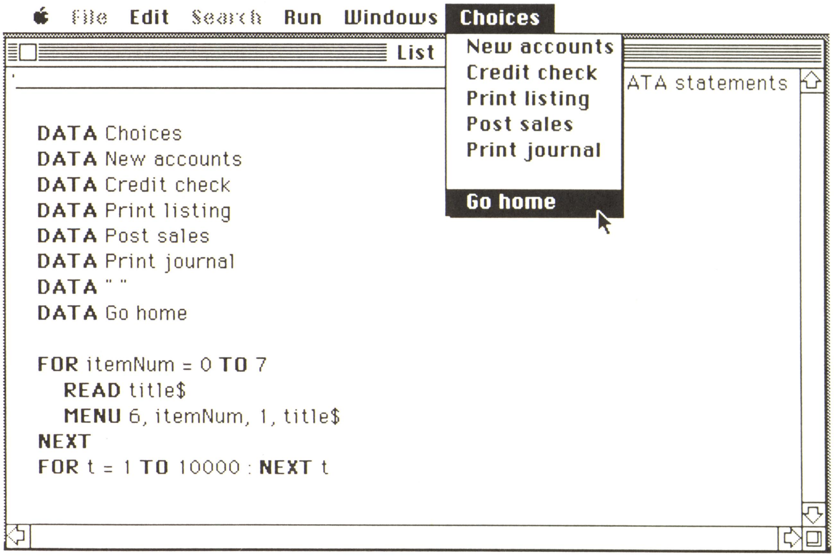
FIGURE 12-12

Creating a long menu with FOR…NEXT

Though you might think at first that a menu like this is useless, there may come a time when you'll want something just like it for listing a group of related choices (like the font sizes in MacWrite for example).

Using DATA and READ to create menus

Let's take a look at another kind of menu, created with the help of two BASIC statements we know and love: DATA and READ. Figure 12-13 shows a clever way of storing the titles for our menus in DATA statements.


File
Edit
Search
Run
Windows
Choices
List
ATA statements
New accounts
Credit check
Print listing
Post sales
Print journal
Go home

```
DATA Choices
DATA New accounts
DATA Credit check
DATA Print listing
DATA Post sales
DATA Print journal
DATA " "
DATA Go home

FOR itemNum = 0 TO 7
   READ title$
   MENU 6, itemNum, 1, title$
NEXT
FOR t = 1 TO 10000 : NEXT t
```

FIGURE 12-13

Using DATA statements to create menus

Then, if we want to change a menu we need only change the information in the DATA statements, and not the MENU statements themselves. Observe that this is not a complete program, since there is no idle loop or menu-choice processing section. This is only the initialization section that will create the menu using the DATA statements.

This program also uses a FOR...NEXT loop to help create the menu, but this time there's another BASIC statement included inside the loop: *READ title$*. This statement gets the value for the variable *title$* from the DATA statements.

You'll notice that one of the DATA statements contains only a blank between quotes. This produces the blank line on the menu between *Print journal* and *Go home*. This helps us organize our menus into groups of operations.

Using the DATA and READ statements to help create our menus is one way of making the menus more flexible. Something rather inflexible we've been doing, though, is always assigning the first menu we create the ID number 6. In the next section we'll see what happens when we create a menu using ID numbers from 1 to 5.

Replacing BASIC's menus

Earlier in this chapter we said that it wasn't a good idea to create a menu with an ID number below 6. This is because BASIC's standard menus are numbered 1 through 5. We mentioned that, as a beginning programmer, you'll probably feel more secure seeing our old familiar BASIC menu bar at the top of the screen. But there are times when it's nice to let our menus replace BASIC's menus. Let's see how we can do this.

As you can see in Figure 12-14 on the next page, we've created a new menu with an ID number of 1. This replaces BASIC's number 1 menu (File) with our new menu, which appears just to the right of the Apple menu. We've used some rather lengthy menu titles in this example to justify our need to remove some of the standard menus in order to make room for our own.

Notice the two MENU statements at the bottom of the program listing. These "blank out" BASIC's standard menus 2 and 3, the Edit and Search menus. These menus are no longer available, as you can see by looking at the menu bar. We have not disturbed the Run and Windows menus, which still appear on the menu bar (although far to the right of their normal positions) and will operate normally.

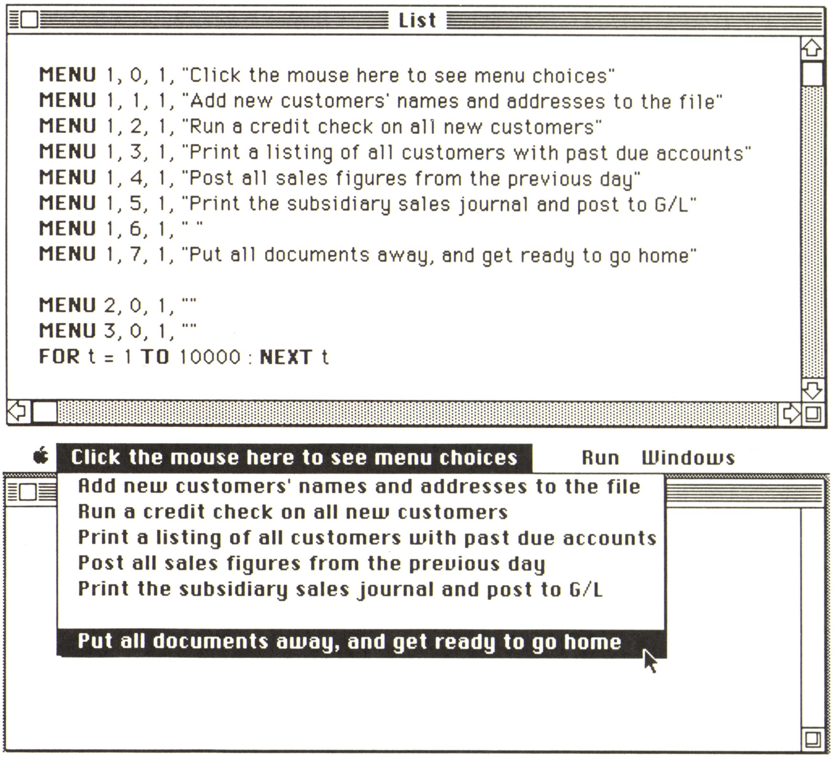


FIGURE 12-14

Replacing BASIC's menus

```
MENU 1, 0, 1, "Click the mouse here to see menu choices"
MENU 1, 1, 1, "Add new customers' names and addresses to the file"
MENU 1, 2, 1, "Run a credit check on all new customers"
MENU 1, 3, 1, "Print a listing of all customers with past due accounts"
MENU 1, 4, 1, "Post all sales figures from the previous day"
MENU 1, 5, 1, "Print the subsidiary sales journal and post to G/L"
MENU 1, 6, 1, " "
MENU 1, 7, 1, "Put all documents away, and get ready to go home"

MENU 2, 0, 1, ""
MENU 3, 0, 1, ""
FOR t = 1 TO 10000 : NEXT t
```

List
Click the mouse here to see menu choices
Run
Windows
Add new customers' names and addresses to the file
Run a credit check on all new customers
Print a listing of all customers with past due accounts
Post all sales figures from the previous day
Print the subsidiary sales journal and post to G/L
Put all documents away, and get ready to go home

Even though our new menu is a bit verbose, it will still operate like any other menu. We can make choices from it, which will be reported by the MENU function. This type of "descriptive" menu is especially useful in programs designed for absolute beginners, who can often use lots of guidance when running a program.

PUTTING CHECK MARKS ON A MENU

Now that we've seen some alternative methods of creating menus, let's go on to explore some additional menu options that give feedback to the user about the status of the menus. We mentioned these earlier, when we were working with our *Shapes* program, so let's find out how to check mark and dim our menu items.

Recall that our program displays some graphics and text on the screen in one of two sizes: large or small. But as the program stands, there is no way of telling which size is currently in use. What we would like is to have a check mark appear next to the menu item that is *currently selected,* as shown in Figure 12-15.

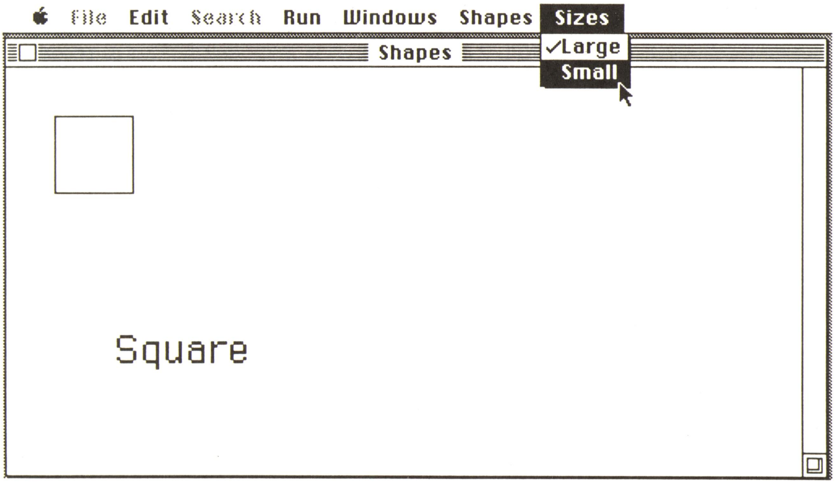


FIGURE 12-15

A check mark on a menu

As you can see in the figure, our Sizes menu takes on another dimension of meaning by having a simple check mark next to the most recent menu choice. Now we know that when we make a choice from the Shapes menu, whatever we choose will be drawn with the chosen size. The program that sets this up is shown in Figure 12-16 on page 274.

This program is an extension of our earlier example, with four additional MENU statements added in the subroutine labeled *Sizes* near the end of the program. The first two MENU statements follow the *Large* label. Notice the third parameter of these statements. Remember, the third parameter of the MENU statement is the *Status* of the menu item. Remember the three different states for a menu item:

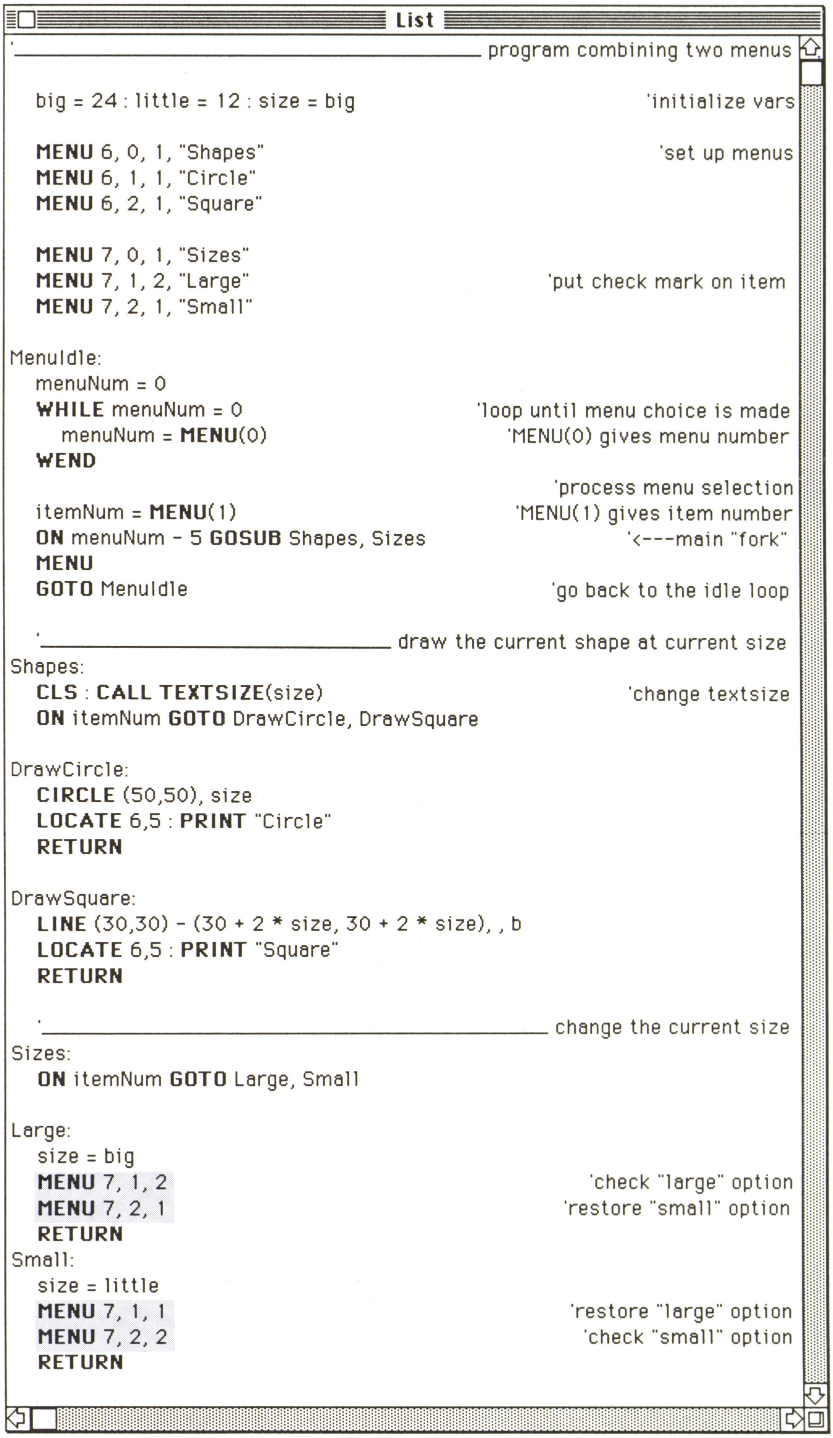
0—Dimmed (item cannot be chosen)
1—Normal (item can be chosen)
2—Check mark preceding selected item

In our program, *Item* number 1 (Large) is assigned a *Status* of 2. This is what makes the check mark appear next to the Large choice on the menu. The second MENU statement simply assigns the Small choice (*Item* parameter 2) a *Status* value of 1, specifying a normal menu item.

In the section following the label *Small,* a similar pair of MENU statements does just the opposite. These statements restore Large to normal status, and put the check mark beside the Small choice.

FIGURE 12-16

A program that puts check marks on a menu



```
'____________________________________ program combining two menus

  big = 24 : little = 12 : size = big                  'initialize vars

  MENU 6, 0, 1, "Shapes"                                  'set up menus
  MENU 6, 1, 1, "Circle"
  MENU 6, 2, 1, "Square"

  MENU 7, 0, 1, "Sizes"
  MENU 7, 1, 2, "Large"                           'put check mark on item
  MENU 7, 2, 1, "Small"

MenuIdle:
  menuNum = 0
  WHILE menuNum = 0                      'loop until menu choice is made
    menuNum = MENU(0)                       'MENU(0) gives menu number
  WEND
                                              'process menu selection
  itemNum = MENU(1)                           'MENU(1) gives item number
  ON menuNum - 5 GOSUB Shapes, Sizes                   '<---main "fork"
  MENU
  GOTO MenuIdle                               'go back to the idle loop

  '____________________________ draw the current shape at current size
Shapes:
  CLS : CALL TEXTSIZE(size)                            'change textsize
  ON itemNum GOTO DrawCircle, DrawSquare

DrawCircle:
  CIRCLE (50,50), size
  LOCATE 6,5 : PRINT "Circle"
  RETURN

DrawSquare:
  LINE (30,30) - (30 + 2 * size, 30 + 2 * size), , b
  LOCATE 6,5 : PRINT "Square"
  RETURN

  '________________________________________ change the current size
Sizes:
  ON itemNum GOTO Large, Small

Large:
  size = big
  MENU 7, 1, 2                                  'check "large" option
  MENU 7, 2, 1                                'restore "small" option
  RETURN
Small:
  size = little
  MENU 7, 1, 1                                'restore "large" option
  MENU 7, 2, 2                                  'check "small" option
  RETURN
```

If you try running this program, you'll see that it's much easier to operate now that we can see which size is currently selected. To really make our program work properly, there's one more statement that we need to change.

Look back to the beginning of the program. Remember that we decided to set the default *size* to *big* so that the user can draw right away, without having to choose a textsize. To be consistent, we should change the MENU statement that originally creates the menu, so that it puts a check mark next to the Large choice. The statement *MENU 7, 1, 2, "Large"* takes care of this.

We've just about learned all there is to learn about menus and the polling approach, except for one thing. From time to time, we've seen menu items that appear dim and, when we try to choose them, nothing happens. Let's see how we can control this effect in our BASIC programs.

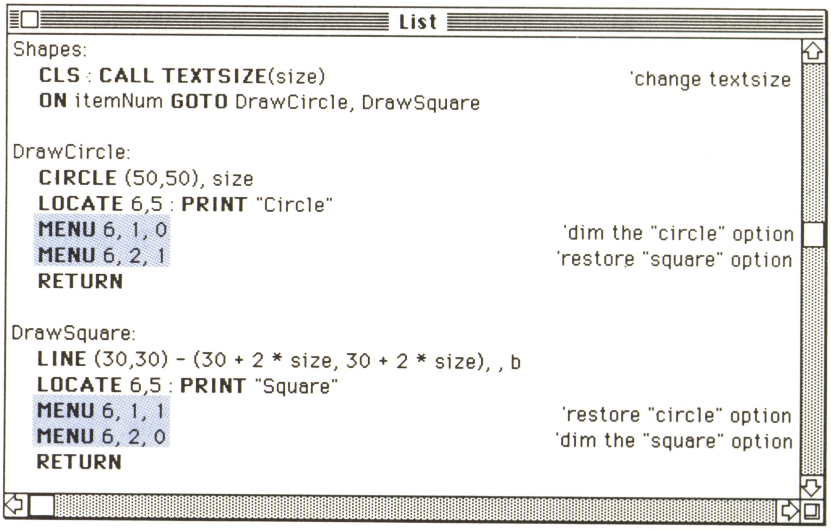
DIMMING A MENU ITEM

Depending upon the particular operation that our program accomplishes, there may be times when we don't want the user to be able to make a particular menu choice. We see examples of this in the menus of BASIC. For example, we can't choose Save from the File menu if we haven't entered a program. The Save item is *dimmed* and does not become highlighted when we try to choose it with the mouse. Let's add this feature to our demonstration program so that we are required to alternate back and forth between drawing circles and squares.

Our example program in Figure 12-17 on the next page is almost the same as the check-mark version we just created. However, some additional MENU statements have been inserted in the *DrawCircle* and *DrawSquare* subroutines that actually do the drawing. The first two MENU statements are in the routine *DrawCircle.* The first statement dims the Circle menu item by assigning it a *Status* of 0. The second statement assigns a *Status* value of 1 to the Square menu item, thereby making it a normal menu item. Similarly, two other MENU statements in the *DrawSquare* routine change the *Status* of the menu items so that the Square menu item is dimmed and the Circle item is normal.

FIGURE 12-17

A program that dims menu items



```
Shapes:
   CLS : CALL TEXTSIZE(size)                              'change textsize
   ON itemNum GOTO DrawCircle, DrawSquare

DrawCircle:
   CIRCLE (50,50), size
   LOCATE 6,5 : PRINT "Circle"
   MENU 6, 1, 0                                      'dim the "circle" option
   MENU 6, 2, 1                                    'restore "square" option
   RETURN

DrawSquare:
   LINE (30,30) - (30 + 2 * size, 30 + 2 * size), , b
   LOCATE 6,5 : PRINT "Square"
   MENU 6, 1, 1                                    'restore "circle" option
   MENU 6, 2, 0                                      'dim the "square" option
   RETURN
```

Figure 12-18 shows what the Shapes menu looks like right after the circle has been drawn. We can see that the Circle menu item has been dimmed, and that the Square menu item still appears normally on the menu and may be chosen.

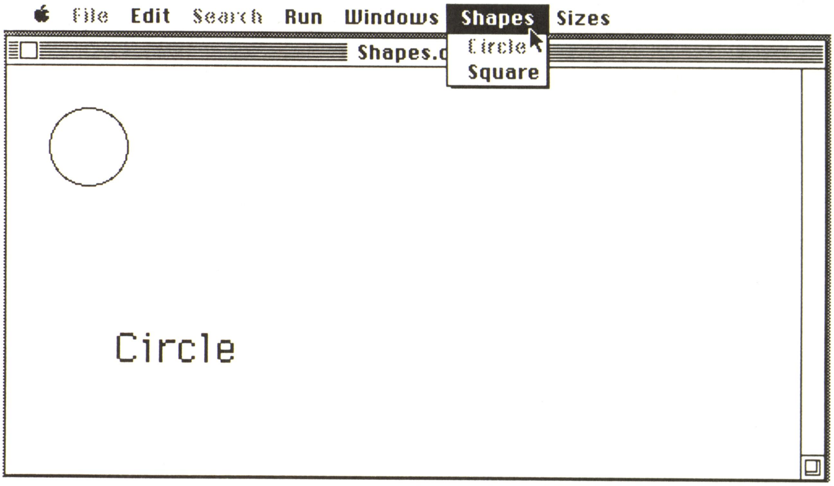

File
Edit
Search
Run
Windows
Shapes
Sizes
Circle
Square
Shapes.c
Circle

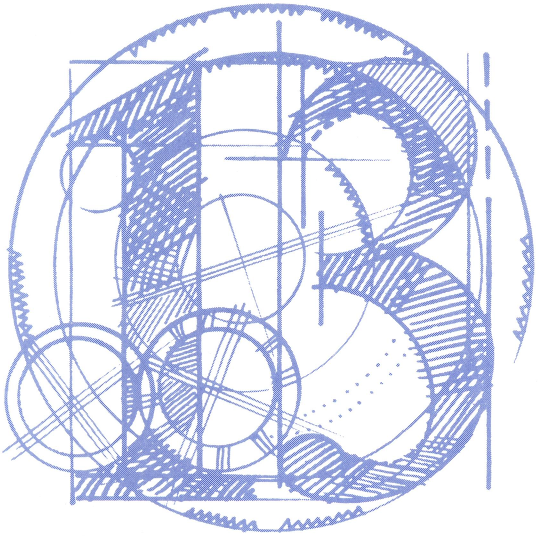
FIGURE 12-18

A dimmed menu item

REFRESH-YOURSELF WINDOW

In this chapter we've learned how to create menus on BASIC's menu bar and use them in our programs. We've seen how this important feature of Microsoft BASIC easily provides a "user-friendly" procedure for making choices about what the program is supposed to do. The example program we've been working with in this chapter can be used as a "shell" for any program you might wish to write that uses two menus.

In the next few chapters, we're going to discuss more of these Macintosh features that make programming and using the Mac a joy. We'll begin with a look at *buttons*—another part of the user interface.



Buttons

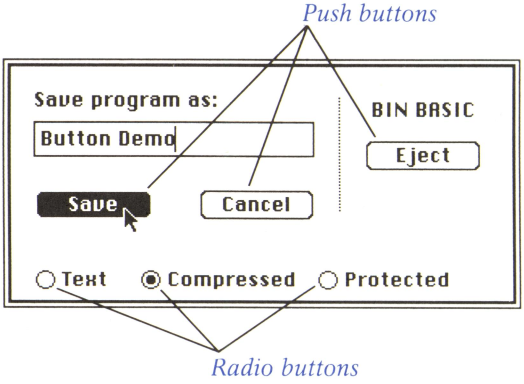
One feature of the Macintosh that sets it apart from many other personal computers is the ability to use the mouse to control the action of a program. We've seen several ways to do this in our BASIC programs. We can use the MOUSE function to tell us where the mouse pointer is and the status of the button. We can use the MENU statement and function to create our own menus, and then operate them with the mouse just like all Macintosh menus. In this chapter, we're going to learn to use yet another mouse-related concept: the *button.*

Using buttons simplifies the operation of our programs from the user's viewpoint. There is less chance of error if the user makes selections by clicking a button with the mouse pointer, rather than entering something from the keyboard.

WHAT IS A BUTTON?

If you've been using the Macintosh to test some of our programs, then you've been using buttons all along. One example is the Save dialog box, shown in Figure 13-1, which contains several buttons. We know from our experience with the Save dialog box that we *give directions* to a program when we click one of these buttons by placing the pointer inside it and pressing the mouse button. As we learned in Chapter 3, we can *Save* the program, *Cancel* the save procedure, *Eject* the disk, or have the program saved in either the *Text, Compressed,* or *Protected* mode. As the Save dialog box exemplifies, buttons allow us to direct our programs to *perform an action* and to *make choices* about how an action will be performed.

There are two different types of buttons in the Save dialog box, which are used for different purposes. Let's discuss each of them now.


Save program as: Button Demo

○ Text ● Compressed ○ Protected

FIGURE 13-1

The Save dialog box and its buttons

Push buttons make things happen

The larger rectangular buttons are called *push buttons.* In our figure, you'll notice that the Save button is highlighted, indicating that the mouse button is being pressed while the pointer is touching the button. This causes the BASIC program (titled *Button Demo* in this case) to be saved on the disk.

Radio buttons indicate selections

Before we save a program using the Save dialog box, there are some options we can select. We tell BASIC our selections using the *radio buttons.* These are the three small round buttons located at the bottom of the dialog box. They are called *radio buttons* because they resemble the buttons used to select the station on a radio.

Here again, we use the mouse to "press" the radio button of our choice. When we click a radio button, a solid black circle appears inside the button, indicating that it has been selected.

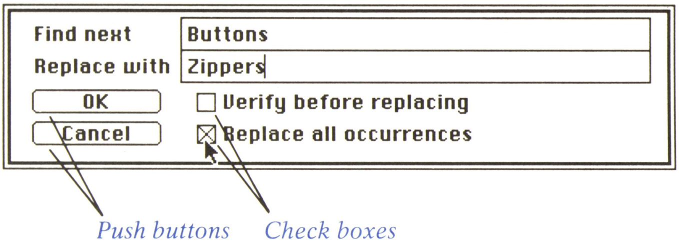
Buttons are user friendly

As we can see from just this one example, buttons make operating the Macintosh very simple. We don't have to memorize complicated commands and then type them perfectly to get things to happen. Instead, the buttons can be used in a "control panel" from which the user can operate the program. Let's take a look at two more examples of buttons that we know and love.

Check boxes in the Search dialog box

The Search dialog box is displayed by BASIC whenever we choose Replace from the Search menu. This is a convenient way to make global changes in our programs. The example in Figure 13-2 would search a program for any occurrence of *Buttons* and replace it with *Zippers.*

There's a new type of button in this dialog box, called the *check box.* While it has its own distinctive appearance, it is used just like a radio button to select an option. In this case, there are two options that can be selected using the check boxes: We can tell BASIC to *Verify before replacing,* which means we'll have to approve every replacement before BASIC


Find next: Buttons
Replace with: Zippers

☐ Verify before replacing
☒ Replace all occurrences

FIGURE 13-2

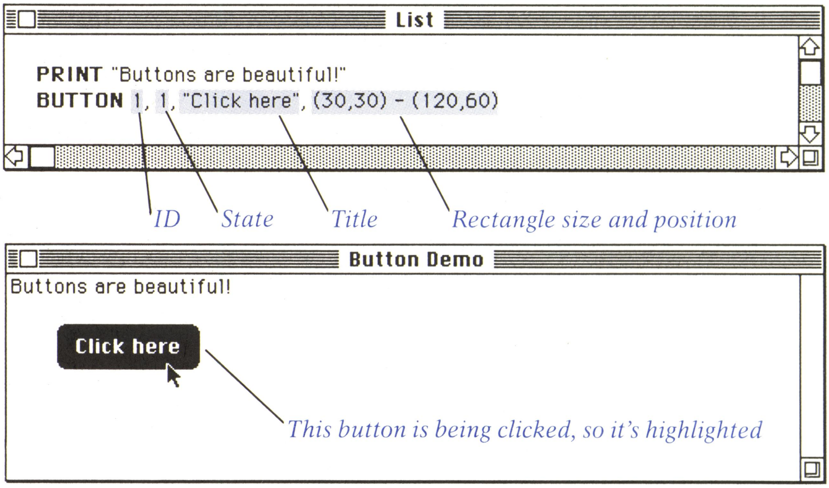
The buttons in the Search dialog box include check boxes will do it. Also, we can tell BASIC to *Replace all occurrences,* which means it will search through our entire program and make the swaps without asking our permission each time. The choice of which button to use, radio or check, is chiefly one of aesthetics. You'll find programs that use one or the other—or both—to make choices.

There are also two push buttons in the Search dialog box. Remember, these are used to *indicate actions* in our program. In this case, there are two things that can happen. If we click the OK button, BASIC goes ahead with the replacement operation. However, if we click the Cancel button, BASIC discontinues the replacement operation.

Now we've seen examples of three different types of buttons used in BASIC. We use the push buttons to have our program perform an action, and use the radio buttons and check boxes to select options. All three of these buttons can be created by our BASIC programs, and they will operate just like all the buttons we've seen here. Let's find out how we can create a button.

CREATING A BUTTON: THE BUTTON STATEMENT

Creating a button in our BASIC programs is very similar to the way we created menus in the previous chapter. The difference is that only one BUTTON statement is needed to create a button, whereas two or more MENU statements are necessary to create a menu. A simple example of how the BUTTON statement can be used is shown in Figure 13-3.



```
PRINT "Buttons are beautiful!"
BUTTON 1, 1, "Click here", (30,30) - (120,60)
```

ID
State
Title
Rectangle size and position
Button Demo

```
Buttons are beautiful!
```

Click here
This button is being clicked, so it's highlighted

FIGURE 13-3

Using the BUTTON statement to create a button

This example program contains only two statements. A PRINT statement displays a message on the screen. Next, a single BUTTON statement creates the push button titled Click Here. You'll notice that the button is highlighted to indicate that it is being clicked. We'll take a closer look at the syntax of the BUTTON statement in a moment, but first, let's learn how buttons work.

All buttons work alike

We've seen several examples of buttons that BASIC uses to control various operations like saving programs, searching, and printing. The buttons in all of these examples look and work similarly. That's because buttons are a standard feature on the Macintosh, and their appearance and operating characteristics are built into the Mac.

The buttons that we create in our BASIC programs are no different. The BUTTON statement causes a button image to automatically appear on the screen in our desired location. Once displayed, the button functions like all other Mac buttons. When you point to it and press the mouse button, the button is highlighted just as if you "pressed" it. (We need not concern ourselves with this feature; it is handled for us automatically by BASIC and the Macintosh.) We *do* need to concern ourselves with where a button is to appear, what type of button it is, its title, and finally, when it is clicked.

Now let's examine the BUTTON statement itself and see how to use it in our programs. The general syntax of the BUTTON statement is:

BUTTON *ID, State, Title, Rectangle, Type*

Let's take each of the parameters for the BUTTON statement in order.

The ID parameter

Although our example program in Figure 13-3 only creates one button, several buttons can be created and used by the same BASIC program. In order for BASIC to keep track of which button is which, each button is assigned an *ID* number, just as menu items are assigned *ID* numbers. Since we only have one button in our program, we've assigned it an *ID* number of 1. If we wanted a second button in the program we would assign it an *ID* number of 2, and so on. As we'll see, the *ID* number will be used later by BASIC to determine which button has been clicked.

The State parameter

You may have noticed that buttons sometimes change their appearance. For example, when a particular button's function is not appropriate at some point, the button is "dimmed" to indicate that it is not available to be clicked. We've also seen that radio buttons can have a black circle in them to indicate that they have been selected, and that check boxes have cross marks in them to indicate that they have been selected.

These variations in button appearance are controlled by the *State* parameter of the BUTTON statement. Here are the values corresponding to the three possible states that a button can have:

0—Dimmed (button cannot be clicked)
1—Normal (button can be clicked)
2—Selected (button was just clicked)

You'll notice that the button we've created in our example program has been assigned a *State* of 1, making it a normal button that can be clicked. As we learn more about using buttons, we'll see how to use the *State* parameter to change the appearance of a button on the screen.

The Title parameter

The *Title* parameter of the BUTTON statement is a string that is used to print a button description. The titles for push buttons automatically appear centered inside the button itself. As we'll see, we must make sure to specify the button's size as large enough for the title to fit inside. Titles for radio buttons and check boxes always appear to the right of the button on the screen.

The BUTTON statement expects the *Title* parameter to be a string expression. In most cases, a string constant is the easiest way to title a button, such as the *Click here* string in our example.

The Rectangle parameter

Rectangle is perhaps the most complicated of the BUTTON statement's parameters. It specifies both *where* the button will appear on the screen, and also its *size*. This is done using the familiar pairs of (x,y) coordinates we use in the LINE statement when we want to draw a box. The first pair of coordinates specifies the position of the upper left corner of the button, and the second pair specifies the lower right corner of the button.

Push buttons can be made any size we want. They will always appear as round-cornered rectangles on the screen, but you can make them really small or really large, depending upon your needs. Usually, it's enough to specify a push button just large enough to contain its title.

On the other hand, radio buttons and check boxes always appear the same size and shape regardless of how large a rectangle is specified.

The main consideration with these buttons is that the rectangle must be wide enough to contain the button and its title, which will be displayed just to the right of the button.

The Type parameter

The last parameter of the BUTTON statement is the button *Type*. This parameter is optional, and you'll notice that it does not appear in the button statement shown in our example program. If you don't specify a particular type of button, BASIC assumes that you want a push button. The values for the three different types of buttons that we can create are:

1—Push button
2—Check box
3—Radio button

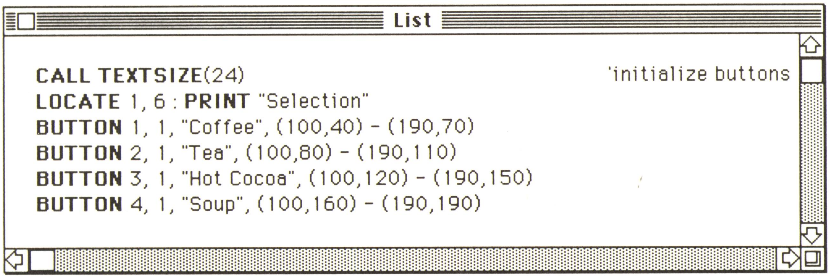
We'll be using all three button types in the examples in this chapter. For now, though, we'll stay with the default push button and see what can be done with it alone.

BUTTON STATEMENT EXAMPLES

There are many different situations where buttons are an ideal way of getting information from the user. We have seen how easy it is to create one button, but what if we need more than one button in our program?

Creating multiple push buttons

We can create as many buttons as we want in our BASIC programs. Remember, we only have to make sure that they all have unique *ID* numbers. The example program in Figure 13-4 creates the four push buttons shown at the top of the next page.



```
CALL TEXTSIZE(24)                                    'initialize buttons
LOCATE 1, 6 : PRINT "Selection"
BUTTON 1, 1, "Coffee", (100,40) - (190,70)
BUTTON 2, 1, "Tea", (100,80) - (190,110)
BUTTON 3, 1, "Hot Cocoa", (100,120) - (190,150)
BUTTON 4, 1, "Soup", (100,160) - (190,190)
```

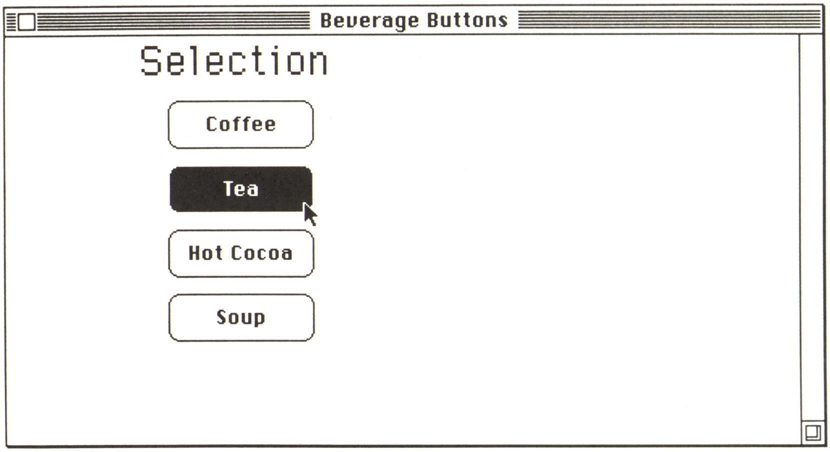
FIGURE 13-4

Creating multiple buttons

(continued)

FIGURE 13-4

(continued)



The first two program lines simply display the heading, *Selection,* in 24-point characters at the top of the screen. Then four BUTTON statements create the four buttons. Notice that the *ID* numbers for the buttons range from 1 to 4. The *State* parameter for all the buttons is 1, indicating a normal, "selectable" button. Again, we've omitted the *Type* parameter from the BUTTON statement to create the default push buttons.

Next, notice the coordinates that specify the rectangle for each of the buttons. Since the buttons are arranged in a vertical column on the screen, the *x coordinates* for all the buttons are the same while the *y coordinates* change from button to button. Also observe that we've specified buttons that are large enough to contain the titles. Until you become familiar with the size of various buttons, you will probably have to experiment with different rectangle coordinates.

If we click one of the buttons now, it automatically becomes highlighted when we hold the mouse button down. This is a visual feedback feature that is automatic and lets the user know the program acknowledges the button-press. Of course, nothing else happens now when a button is clicked because the program ends right after it creates the four buttons. But once created, buttons seem to have a life of their own because they are controlled automatically by Microsoft BASIC.

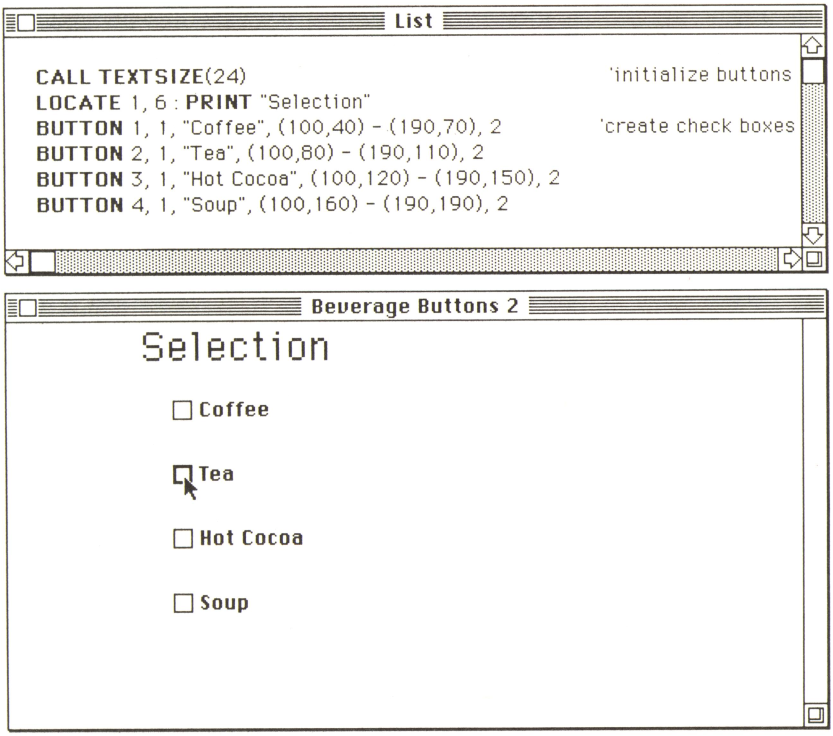
Now let's create the other button types.

Creating check boxes

Just to illustrate how easily we can change a button's type, let's make some slight changes to the program from our last example. The amended program is shown in Figure 13-5.

FIGURE 13-5

Creating check boxes



```
CALL TEXTSIZE(24)                                              'initialize buttons
LOCATE 1, 6 : PRINT "Selection"
BUTTON 1, 1, "Coffee", (100,40) - (190,70), 2                  'create check boxes
BUTTON 2, 1, "Tea", (100,80) - (190,110), 2
BUTTON 3, 1, "Hot Cocoa", (100,120) - (190,150), 2
BUTTON 4, 1, "Soup", (100,160) - (190,190), 2
```

```
Selection

  ☐ Coffee

  ☐ Tea

  ☐ Hot Cocoa

  ☐ Soup
```

We've added *Type* parameter 2 to each of the four BUTTON statements. Everything else is identical to the last program. As you can see, the program now creates check boxes instead of push buttons.

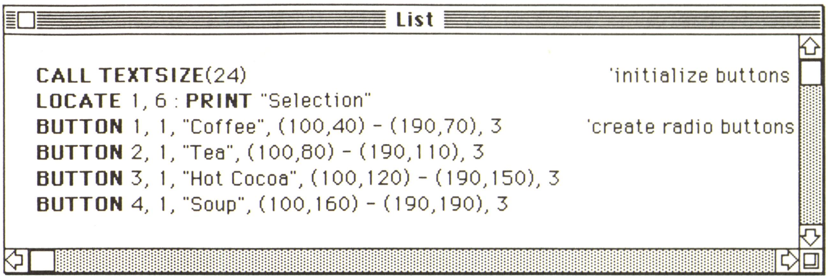
If we now click one of these check boxes, the outline of the box itself darkens while the button is clicked, as shown with the Tea check box in Figure 13-5. But at this stage of our program, no cross mark appears in the box when we release the mouse button. The reason the cross mark doesn't automatically appear is that it's controlled by the *State* parameter, which in turn must be controlled by our program. We'll learn how to do this a bit later. For now, let's make one more small change to our current program.

Creating radio buttons

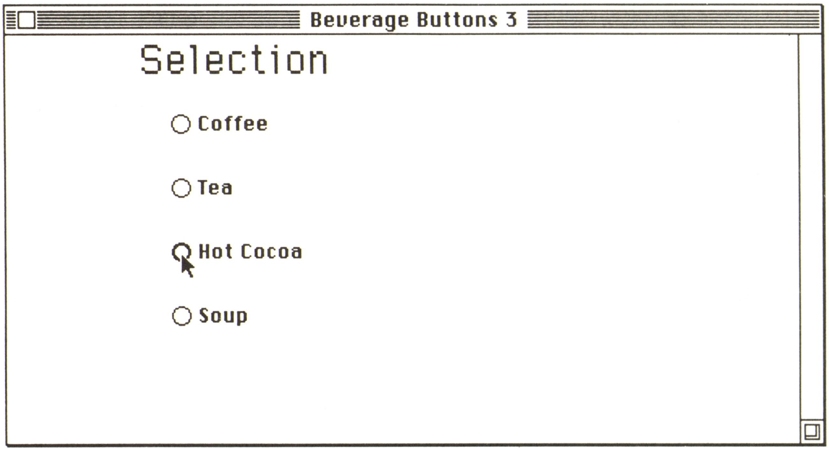
Radio buttons are just as easy to create as push buttons and check boxes: All we have to do is change the *Type* parameter of the BUTTON statement to 3, as shown in Figure 13-6 on the following page.

FIGURE 13-6

Creating radio buttons



```
CALL TEXTSIZE(24)                                          'initialize buttons
LOCATE 1, 6 : PRINT "Selection"
BUTTON 1, 1, "Coffee", (100,40) - (190,70), 3             'create radio buttons
BUTTON 2, 1, "Tea", (100,80) - (190,110), 3
BUTTON 3, 1, "Hot Cocoa", (100,120) - (190,150), 3
BUTTON 4, 1, "Soup", (100,160) - (190,190), 3
```



```
Selection

○ Coffee

○ Tea

○ Hot Cocoa

○ Soup
```

You can see by the Hot Cocoa selection in the figure that, like the check box, the outline of a radio button darkens when it's clicked. But again, when the mouse button is released, the black spot indicating selection does not automatically appear in the radio button. We have to design our programs to insert the selection spot, which we will learn how to do later in the chapter.

So far we've seen that buttons are very intuitive methods for letting the user control the operation of a program. Push buttons make things like saving programs happen, while check boxes and radio buttons allow the user to select options. We now know that buttons are easy to create using the BUTTON statement. But there's more to a button than creating it. Let's forge on now, and see how we can really put buttons to work in our programs.

USING BUTTONS IN PROGRAMS

Once a button has been created using the BUTTON statement, BASIC and the Macintosh work together to make them work for us. When a button is clicked, BASIC automatically knows it and also which button was clicked.

This information is available to our BASIC programs through the DIALOG function, which is the key to using buttons.

The DIALOG function

The Macintosh is such a friendly computer that it considers the exchange of information and commands between itself and the user *dialog*. That's why we always refer to *dialog boxes*, such as the Save dialog box that allows us to save a BASIC program.

There are many different *events* that can occur in a dialog box, and the DIALOG function is used to distinguish between them. Although most of the *dialog events* are not related to buttons, we're going to limit our discussion to button-related dialog events for now. What we're concerned with is that the DIALOG function can answer two important questions we need to know in order to use buttons in our programs:

- Has a button been clicked?
- Which button was clicked?

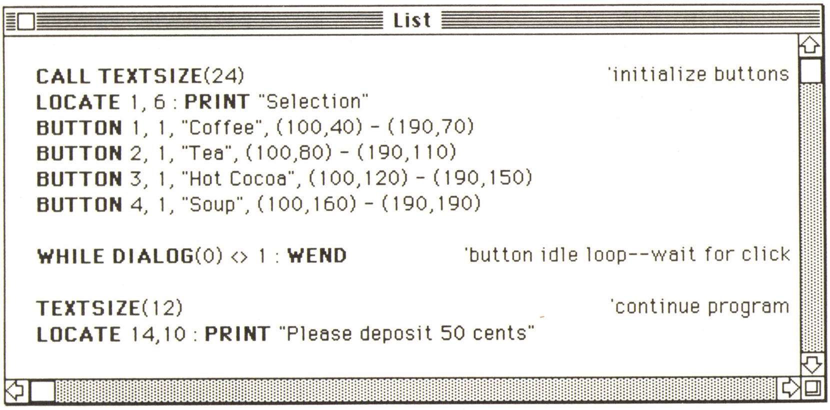
Let's take these questions one at a time, and see how the DIALOG function can respond to them.

DIALOG(0): Has a button been clicked?

We'll continue using the example program that looks like a vending machine. As shown in Figure 13-7, we've reverted to the push-button version of the program, and added button idle and continue sections.

FIGURE 13-7

The DIALOG(0) function after detecting a button click

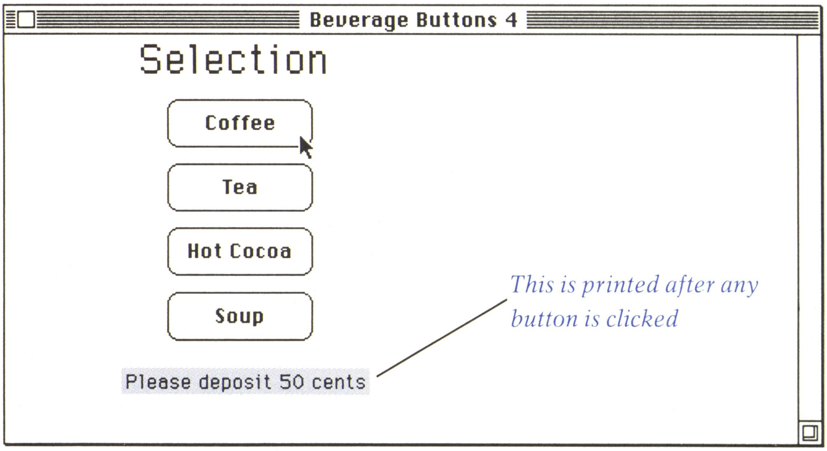


```
CALL TEXTSIZE(24)                                         'initialize buttons
LOCATE 1, 6 : PRINT "Selection"
BUTTON 1, 1, "Coffee", (100,40) - (190,70)
BUTTON 2, 1, "Tea", (100,80) - (190,110)
BUTTON 3, 1, "Hot Cocoa", (100,120) - (190,150)
BUTTON 4, 1, "Soup", (100,160) - (190,190)

WHILE DIALOG(0) <> 1 : WEND               'button idle loop--wait for click

TEXTSIZE(12)                                              'continue program
LOCATE 14,10 : PRINT "Please deposit 50 cents"
```

(continued)

FIGURE 13-7

(continued)



Beverage Buttons 4

Selection

Please deposit 50 cents

This is printed after any button is clicked

The DIALOG function works much like the MENU function we covered in the previous chapter. The DIALOG function has only one argument, which is used to specify what information is desired. To find out if *any* dialog event has occurred, we must first use the DIALOG function with an argument of 0. This can be accomplished with an assignment statement, like this:

```
event = DIALOG(0)
```

The value returned by the DIALOG function tells which dialog event has occurred. At this point, we're only interested in the first two values:

0—Nothing has happened
1—A button was clicked

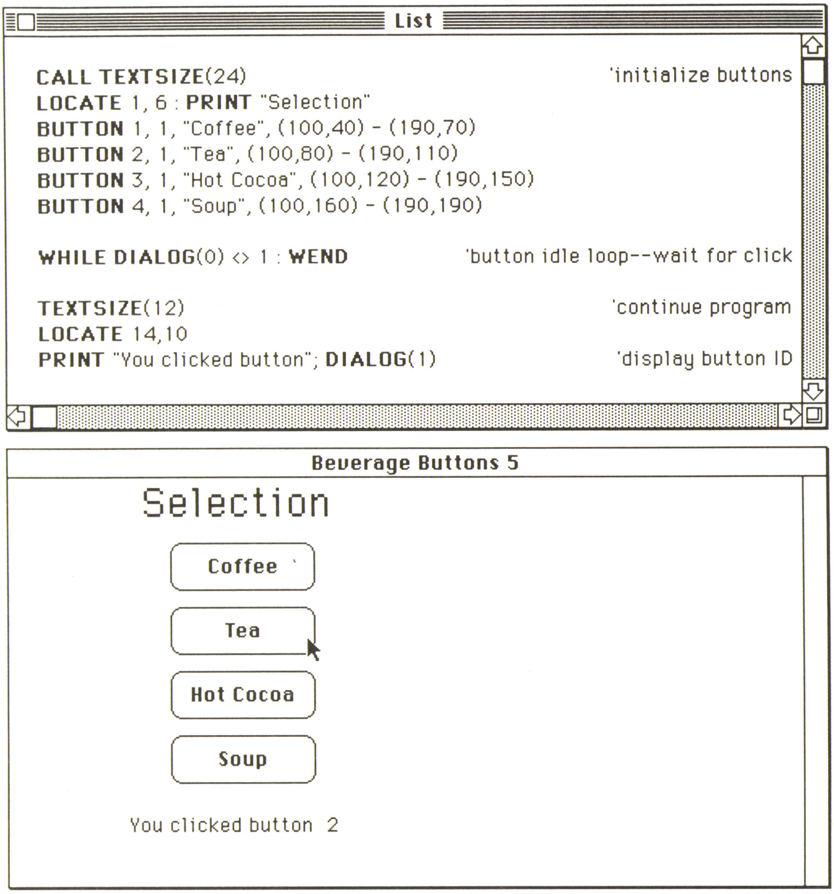
So if a button hasn't been clicked, the DIALOG function returns a value of 0. If a button has been clicked, the function returns a value of 1. That is enough to tell us the answer to the first of our two questions: Has a button been clicked?

The easiest way to have our program look for a button click is to place the DIALOG function inside a WHILE...WEND loop, as shown in Figure 13-7. This makes the program loop until the value returned by the DIALOG function is *equal* to 1, indicating that a button was clicked. When that happens, the program falls out of the loop and continues on to the following statements which display the price.

We've seen that we can use the DIALOG(0) function to find out if a button has been clicked, but we still have no idea *which* button was clicked. When we run this program now, the price message is displayed no matter which button was clicked. Let's now see how we can use the DIALOG function to determine which one was clicked.

DIALOG(1): Which button was clicked?

Our current program creates four buttons and determines when any one of them is clicked. To find out *which* button is clicked, we must use the DIALOG(1) function. This function returns a value that is equal to the *ID* of the button that was just clicked. Let's change the PRINT statement at the end of our last example program to include the DIALOG(1) function, as shown in Figure 13-8.



```
CALL TEXTSIZE(24)                                        'initialize buttons
LOCATE 1, 6 : PRINT "Selection"
BUTTON 1, 1, "Coffee", (100,40) - (190,70)
BUTTON 2, 1, "Tea", (100,80) - (190,110)
BUTTON 3, 1, "Hot Cocoa", (100,120) - (190,150)
BUTTON 4, 1, "Soup", (100,160) - (190,190)

WHILE DIALOG(0) <> 1 : WEND                  'button idle loop--wait for click

TEXTSIZE(12)                                             'continue program
LOCATE 14,10
PRINT "You clicked button"; DIALOG(1)                    'display button ID
```

```
You clicked button  2
```

FIGURE 13-8

Using the DIALOG(1) function to find out which button was clicked

As you can see in the figure, we've included the DIALOG(1) function right in the PRINT statement itself. This prints out the *ID* number of the button that was clicked. Remember, the *ID* number is assigned to a button when it's created by the BUTTON statement. The Tea button was created as button number 2, and that's the number that has been returned by the DIALOG(1) function.

Normally, we want our program to do something meaningful when a button is clicked, like take some action based on the button or buttons selected. When more than one button is involved in a program, it's a good idea to save the button *ID* returned by the DIALOG(1) function in a variable like this:

```
buttonID = DIALOG(1)
```

The button *ID* can then be used by the program to control its operation.

Putting DIALOG(0) and DIALOG(1) together

An important point to understand about the DIALOG functions with buttons is that they must be used together. That is, our program must first use DIALOG(0) to find out if a button has been clicked, and then use DIALOG(1) to find out which button. You can't use DIALOG(1) without first using DIALOG(0).

Well, our program is coming along nicely. We have learned how to create buttons in our programs using the BUTTON statement, and how to use the DIALOG function to find out when a button is clicked, and which button it is. In the next section, we'll learn how to make our programs respond to this information.

What action to take?

We learned early in this chapter that buttons do two things: *Push buttons* make things happen, while *check boxes* and *radio buttons* are used to select options. There are several approaches to writing BASIC programs that use buttons, depending on how complex the program is for the user. In this section, we'll take a look at two popular methods: the IF...THEN statement, which is a simple approach, and the ON...GOSUB, which can handle more structured operations.

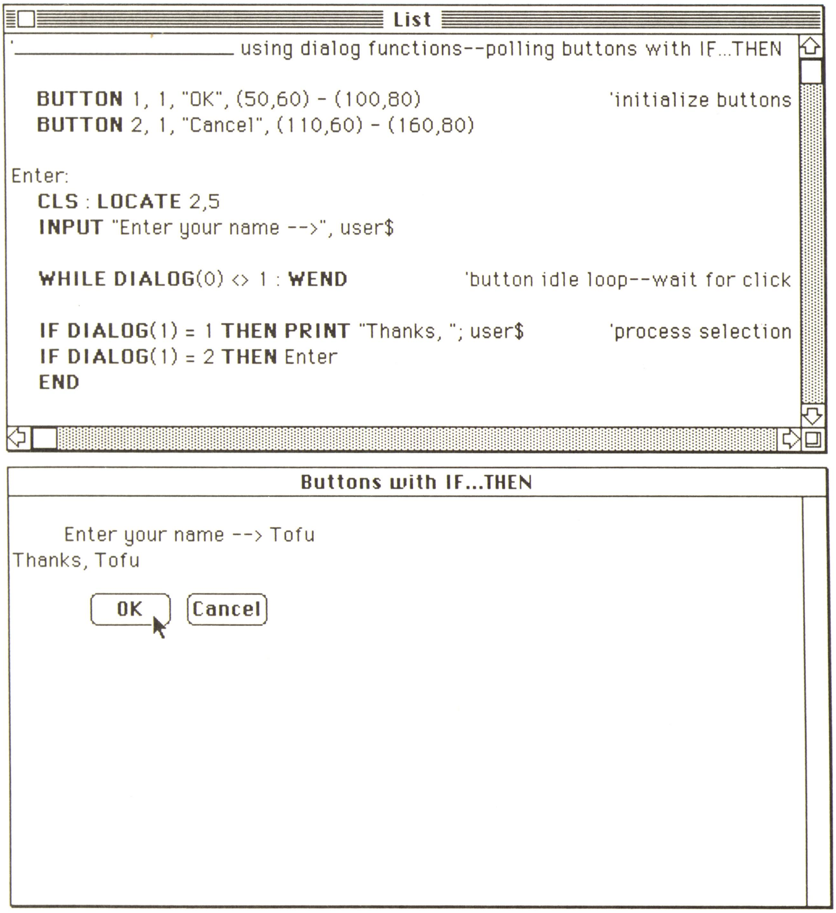
Using IF... THEN with buttons

The easiest way to control a program with a button is to use an IF...THEN statement to test the value returned by the DIALOG(1) function. The example program in Figure 13-9 shows how this works.

Here's a classic button application that looks like something we'd expect to see on the Macintosh. Of course, the program itself does nothing spectacular, except illustrate how easily buttons can be put to work.

The program uses two BUTTON statements to create two buttons titled OK and Cancel, and then prompts the user to *Enter your name.* Once the user types a name and presses the Return key, the program enters the WHILE...WEND "wait loop" to wait for a button to be clicked.

Since there are only two buttons available, it's easy for the IF... THEN statement to find out which one was clicked, and then perform some



```
'________________________ using dialog functions--polling buttons with IF...THEN

   BUTTON 1, 1, "OK", (50,60) - (100,80)                      'initialize buttons
   BUTTON 2, 1, "Cancel", (110,60) - (160,80)

Enter:
   CLS : LOCATE 2,5
   INPUT "Enter your name -->", user$

   WHILE DIALOG(0) <> 1 : WEND                  'button idle loop--wait for click

   IF DIALOG(1) = 1 THEN PRINT "Thanks, "; user$               'process selection
   IF DIALOG(1) = 2 THEN Enter
   END
```

```
    Enter your name --> Tofu
Thanks, Tofu
```

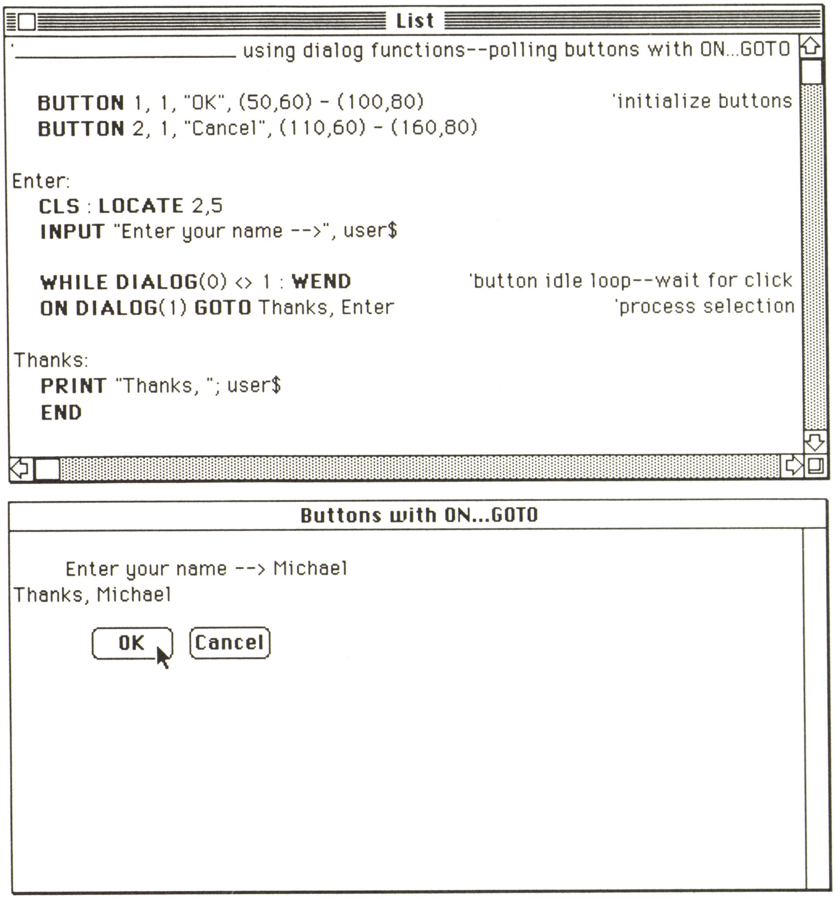
FIGURE 13-9

Using IF... THEN statements to put buttons to work operation. The first IF...THEN statement checks to see if the value returned by DIALOG(1) is equal to 1, indicating that button number 1 (OK) was clicked. If it was, the program prints *Thank you* and the name entered. The second IF...THEN statement checks to see if button number 2 (Cancel) was clicked, and if so, sends the program back to the label *Enter* to start the name entry over.

This method works fine when our program doesn't have to do much when a button is clicked. But most of the time we'll want to use the next method, which provides more flexibility.

Using ON...GOTO and ON...GOSUB with buttons

To keep our attention focused on the use of the ON...GOTO and ON...GOSUB statements, we're going to simply alter the processing section of our previous example program. See Figure 13-10.



```
'_______________ using dialog functions--polling buttons with ON...GOTO

  BUTTON 1, 1, "OK", (50,60) - (100,80)                    'initialize buttons
  BUTTON 2, 1, "Cancel", (110,60) - (160,80)

Enter:
  CLS : LOCATE 2,5
  INPUT "Enter your name -->", user$

  WHILE DIALOG(0) <> 1 : WEND            'button idle loop--wait for click
  ON DIALOG(1) GOTO Thanks, Enter                          'process selection

Thanks:
  PRINT "Thanks, "; user$
  END
```

```
    Enter your name --> Michael
Thanks, Michael
```

FIGURE 13-10

Using ON...GOTO with buttons

Remember, when our program leaves the WHILE...WEND loop it knows that a button has been clicked. We can then use the DIALOG(1) function directly, as the index of the ON...GOTO statement. This statement directs the program to either the *Thanks* label, or back to the *Enter* label, depending on the button *ID* number returned by DIALOG(1).

You can see that it would be a simple matter to add more buttons if we needed to, and then just add more labels to the ON...GOTO statement to provide the program branch. Also, each label can mark the beginning of a block of BASIC statements to be executed when the button is clicked. When the IF...THEN statement is used, as in the previous example, it's not practical to have more than one or two statements following the THEN.

Although we won't show you an example here, we could replace the ON...GOTO statement with an ON...GOSUB statement if we wanted to use subroutines to perform the various actions indicated by the buttons. We'll be using this approach in some of the future example programs later on in this chapter.

PROGRAM STRUCTURE WITH BUTTONS

No doubt you've noticed that the example programs we've used in this chapter have similar characteristics. That's because the nature of buttons recommends a certain program structure in order to use them most effectively. Figure 13-11 shows a diagram of this structure, often called the *polling approach.*

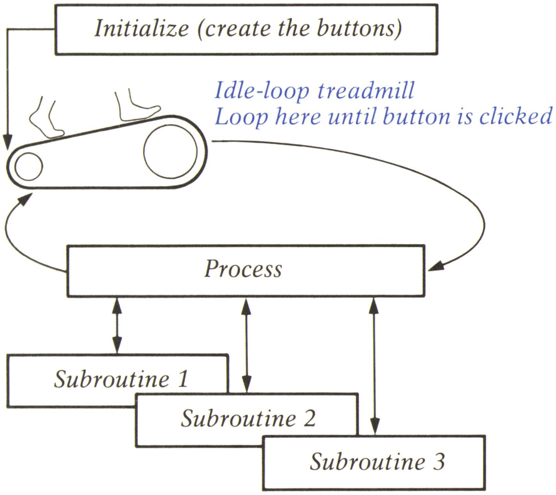

Initialize (create the buttons)
Idle-loop treadmill
Loop here until button is clicked
Process
Subroutine 1
Subroutine 2
Subroutine 3

FIGURE 13-11

The structure of programs using buttons and the polling approach

With this structure, there is always an *initialization* section in which the buttons are created. Then, the program does a little workout on a "treadmill" loop, waiting for the user to click a button with the mouse. Once this happens, the program "falls" out of the treadmill loop and executes statements that process the user's selection.

At this point, we've learned enough to adequately use buttons in our BASIC programs. We can create them with the BUTTON statement, then use the DIALOG function to tell when a button is pressed and which one it was. We've also learned how IF...THEN, ON...GOTO, and ON... GOSUB can use this information to direct our program's operation. To add still more tools to our button workbench, we will now learn about some of the optional features we can use with buttons.

BUTTON OPTIONS

As we mentioned earlier, buttons can sometimes be dimmed, which means they can't be pressed. Also, we've noticed that check boxes and radio buttons show their status with either a cross mark or a black dot. These are optional features that we can use with the buttons we create in our BASIC programs. In this section we'll learn how to dim a button so it can't be clicked, and also how to put cross marks in check boxes and black dots in radio buttons.

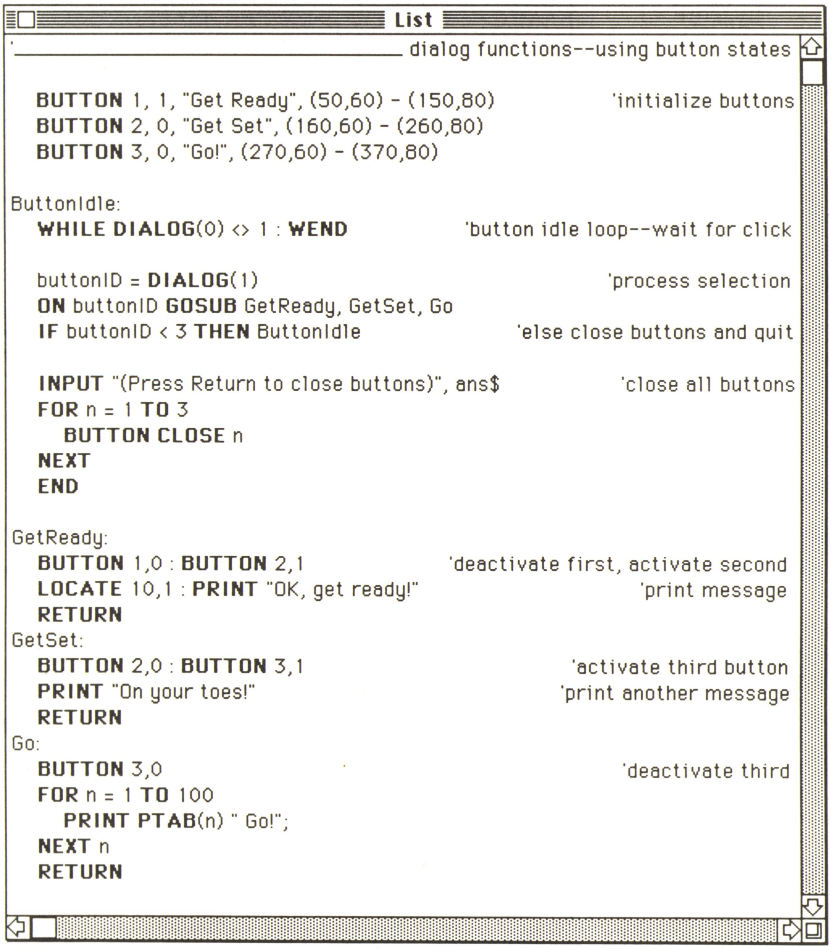
Remember that the BUTTON statement uses a *State* parameter to indicate what state the button will be in, and that there are three different states for a button:

0—Dimmed and unselectable
1—Normal and selectable
2—Selected check box and radio button only

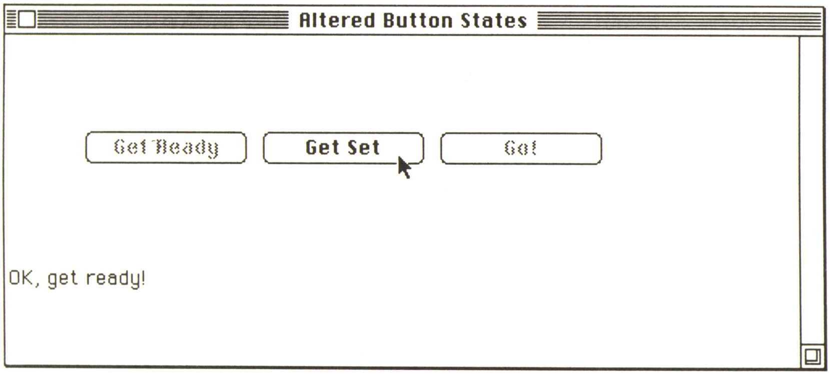
The dimmed and normal states can apply to all three types of buttons. For various reasons, we might want to "disable" certain buttons at different times during the execution of our program by dimming them. The selected state only applies to check boxes and radio buttons, which are the only buttons that can be displayed as selected.

Dimming an unselectable button

There are times during the execution of a program when the action that would be invoked by pressing a certain button would be inappropriate. This concept is similar to the idea of menu items being unavailable (for example, we've seen that when a program is running, the Search menu is dimmed) and is dealt with in almost the same way. The program and output in Figure 13-12 illustrate this feature.



```
'_______________________________________ dialog functions--using button states

   BUTTON 1, 1, "Get Ready", (50,60) - (150,80)              'initialize buttons
   BUTTON 2, 0, "Get Set", (160,60) - (260,80)
   BUTTON 3, 0, "Go!", (270,60) - (370,80)

ButtonIdle:
   WHILE DIALOG(0) <> 1 : WEND                  'button idle loop--wait for click

   buttonID = DIALOG(1)                                       'process selection
   ON buttonID GOSUB GetReady, GetSet, Go
   IF buttonID < 3 THEN ButtonIdle                 'else close buttons and quit

   INPUT "(Press Return to close buttons)", ans$              'close all buttons
   FOR n = 1 TO 3
      BUTTON CLOSE n
   NEXT
   END

GetReady:
   BUTTON 1,0 : BUTTON 2,1                    'deactivate first, activate second
   LOCATE 10,1 : PRINT "OK, get ready!"                           'print message
   RETURN
GetSet:
   BUTTON 2,0 : BUTTON 3,1                                 'activate third button
   PRINT "On your toes!"                                 'print another message
   RETURN
Go:
   BUTTON 3,0                                                 'deactivate third
   FOR n = 1 TO 100
      PRINT PTAB(n) " Go!";
   NEXT n
   RETURN
```



```
OK, get ready!
```

FIGURE 13-12

Dimming a button

We've created three buttons, titled *Get Ready, Get Set,* and *Go,* that could control the beginning of a race. This example illustrates a situation that requires three actions to be performed *in sequence,* which often happens in programs. We can control the sequence of operations by using buttons like we have in this example.

The program begins by displaying all three buttons, but only the first button is normal. The other two buttons are dimmed and cannot be clicked. The two conditions are set up by the statements that create the buttons. Notice that the first BUTTON statement has a *State* parameter of 1, which creates a normal button. The other two BUTTON statements, though, have *State* parameters of 0, which create dimmed buttons.

When the Get Ready button is clicked, the program proceeds to the processing section. Here, the DIALOG(1) function assigns the button *ID* number to the variable *buttonID,* which becomes the index of an ON... GOSUB statement. This directs the program to one of three subroutines, depending on which button was clicked.

The subroutines *GetReady, GetSet,* and *Go* could perform any actual task we needed in our program if we inserted BASIC statements that actually performed some process. Right now, though, they contain the additional BUTTON statements that change the state of the buttons, as well as some simple statements that print various messages at the bottom of the Output window.

The subroutine *GetReady* contains two BUTTON statements that only have the button *ID* and *State* parameters. The first BUTTON statement sets the *State* of button number 1 to 0, which makes it dim and unselectable. The second BUTTON statement sets the *State* of button number 2 to 1, making it selectable.

In a similar manner, the other buttons are dimmed and made unselectable as we click them. After we click the Go button, all three buttons will be dimmed.

The BUTTON CLOSE statement

You may have noticed by this time that buttons can be very hardy creatures. We can clear the screen using CLS, and the buttons will remain. We can even close the Output window, and reopen it by choosing Show Output from the Windows menu, and the buttons will be recreated. Maintenance of buttons—their creation, shape, appearance when selected, and so on—is taken care of internally by BASIC, and BASIC assumes that whatever else happens on the screen, they will remain unaffected.

So how do we get rid of buttons once they're created? The following statement does the trick:

BUTTON CLOSE *n*

where *n* is the button *ID* number of the button we want closed. In our program, the section following the comment *close all buttons* closes the three buttons we created. After we have clicked all the buttons in sequence, the message *(Press Return to close buttons)* is displayed. When we press Return, the program proceeds to the FOR...NEXT loop in this section, which contains the command:

```
BUTTON CLOSE n
```

where *n* is each button *ID* number from 1 to 3.

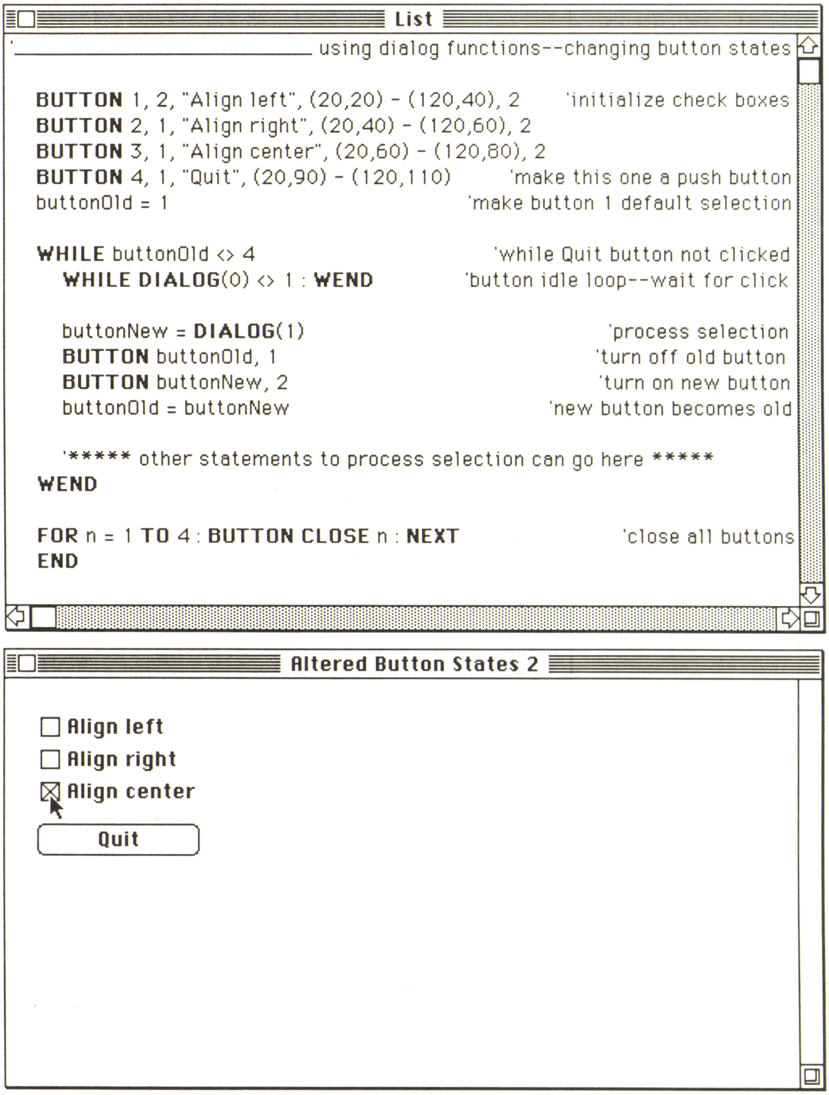
The process of dimming, activating, and closing buttons applies to check boxes and radio buttons as well. Simply use the same approach to change the state of the button using a BUTTON statement that only specifies the button *ID* and *State* parameters.

Showing the selected button

The check boxes and radio buttons we've seen while using the Macintosh have all indicated when they were selected. A selected check box contains a cross mark, and a selected radio button has a black dot in its center. We can create these button elements with BASIC using the *State* parameter of the BUTTON statement.

Remember, check boxes and radio buttons allow the user to make choices, while push buttons are used to make things happen. Consequently, check boxes and radio buttons need to visually indicate the current selection to the user. Let's look at a typical situation where we might want to employ this concept.

Word-processing programs will usually give the user the option to align text either on the left, right, or center of the page. The program in Figure 13-13 creates these options, as well as a push button that ends the program and removes the buttons. We have clicked the center-alignment check box, indicated by the cross mark in the box.



```
'_____________________________ using dialog functions--changing button states

BUTTON 1, 2, "Align left", (20,20) - (120,40), 2       'initialize check boxes
BUTTON 2, 1, "Align right", (20,40) - (120,60), 2
BUTTON 3, 1, "Align center", (20,60) - (120,80), 2
BUTTON 4, 1, "Quit", (20,90) - (120,110)          'make this one a push button
buttonOld = 1                                   'make button 1 default selection

WHILE buttonOld <> 4                                 'while Quit button not clicked
  WHILE DIALOG(0) <> 1 : WEND                   'button idle loop--wait for click

  buttonNew = DIALOG(1)                                          'process selection
  BUTTON buttonOld, 1                                           'turn off old button
  BUTTON buttonNew, 2                                            'turn on new button
  buttonOld = buttonNew                                     'new button becomes old

  '***** other statements to process selection can go here *****
WEND

FOR n = 1 TO 4 : BUTTON CLOSE n : NEXT                          'close all buttons
END
```

☐ Align left
☐ Align right
☒ Align center

FIGURE 13-13

A program to show a cross mark

Our program first creates the three check boxes. You'll notice that check box number 1 is assigned a *State* of 2, which makes that button the default selection. Also notice that we use a variable, *buttonOld,* to keep track of the currently selected button, and it is initialized to 1.

When any of the four buttons is pressed the program enters the processing section. There are four things that must be done here, just to change the selected button. Other, "real" program tasks could also be done here (like whatever it would take to really align the text), but we're not going to complicate things. Here are the four things that must be done:

1. Get the button *ID* using DIALOG(1).
2. Deselect the currently selected button.
3. Select the button that was just clicked.
4. Save the button *ID* for the next button click.

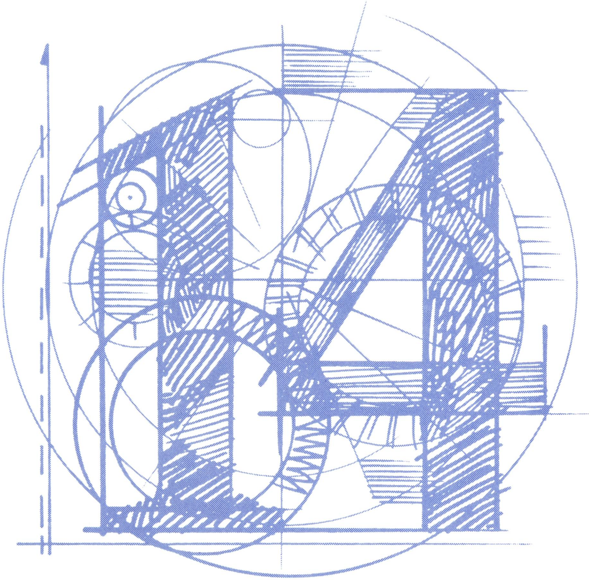
Our program neatly uses one BASIC statement to satisfy each of these four requirements. We set *buttonNew* to the value that was obtained from DIALOG(1). Next, the variables *buttonOld* and *buttonNew* are used in BUTTON statements to turn the buttons on or off. Finally, the new button selection is saved in the variable *buttonOld.*

If we get tired of check boxes, we can always change our program to use radio buttons. All we have to do is change the *Type* parameter of the BUTTON statements that originally create the buttons from 2 to 3. Everything else in the program would be the same as in the previous example. Only the button *Type* would be changed to create a radio button instead of a check box. As you get better at programming, you might want to specify the button *Type* with a variable.

REFRESH-YOURSELF WINDOW

In this chapter, we've learned to use one of the most exciting tools available to our BASIC programs: buttons. These programming tools allow us to create programs that operate the Macintosh as it was intended to be operated. Instead of having to remember and type obscure commands and options, the user of our programs can now use the mouse to press buttons on the screen.

The use of such innovative tools as buttons changes the way we write our BASIC programs. We no longer need complex INPUT statements for user selections because buttons are a simple form of input—"softer software," you might say. Next, we'll look at another tool in our Macintosh user interface: *windows.*



Windows, Dialog Boxes, and Edit Fields

Prior to the advent of the Macintosh, computer programmers had very little to work with in terms of the "human-computer" interface. The user always typed commands and input from the keyboard, and the computer always displayed information on the screen. A typical human-computer dialog from those pre-Macintosh times might look like this:

```
ENTER YOUR NAME? PAUL
ENTER YOUR AGE? 30
WOULD YOU LIKE A HAMBURGER? YES
```

Today, things have changed. When programming the Macintosh using Microsoft BASIC, we must concern ourselves with other forms of "user interface" such as the mouse, menus, and buttons we've learned to work with. In addition, we're no longer limited to using one "screen" to display graphics and other information. Instead, we can present users with up to four different *windows* through which our programs can send and receive information and graphics.

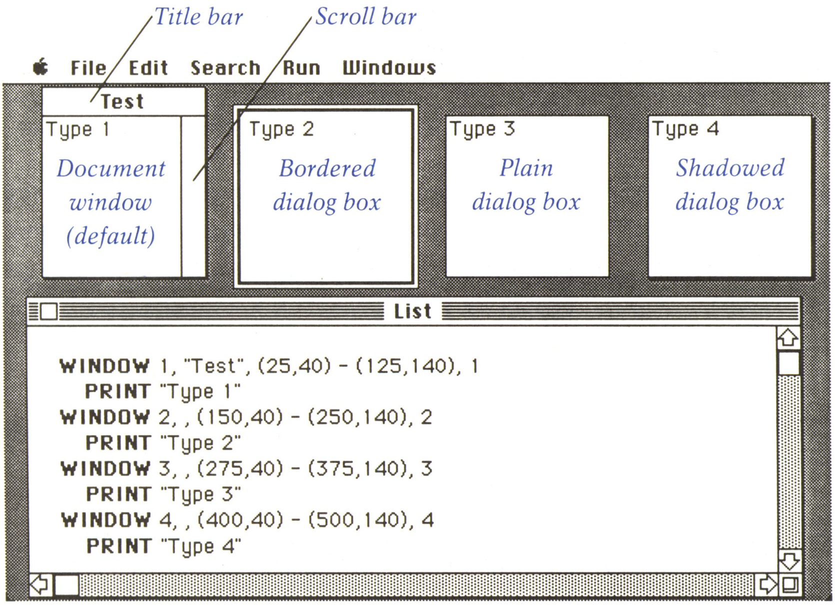
We've benefited from a number of different windows ourselves as we've created our BASIC programs. We've entered our programs in the *List window.* When we've run our programs, we've given them information and they've given us their output in the *Output window.* We're also familiar with the *Command window,* which we've used to enter individual BASIC commands. And although the Save dialog box we worked with in the last chapter is called a *dialog box,* it's actually a type of window that we can create with our programs.

The way windows appear to work on the Macintosh would seem to indicate that they are very complex and difficult to program. The first part is true: They are complex. But they're not difficult to program because the Macintosh does most of the window management work for us. Once we've created a window in our program, we can continue to use the same graphics and text statements that we've been using all along.

In this chapter, we'll learn how to create and use windows in our own programs. We'll find that windows add power and flexibility, as well as aesthetic value to the way our programs "look" when run. We'll also learn how to manage multiple windows in a program, and how to *refresh* a window when it has been covered up. We will also discover *edit fields,* and how to use them with buttons to enter information into windows or dialog boxes. First, let's discover the types of windows we can create.

FOUR TYPES OF WINDOWS

There are four different types of windows that we can use in our BASIC programs, as displayed in Figure 14-1. We'll explain how the WINDOW statement works shortly. For now, just notice the different windows.


Title bar
Scroll bar
File
Edit
Search
Run
Windows
Test
Type 1
Document window (default)
Type 2
Bordered dialog box
Type 3
Plain dialog box
Type 4
Shadowed dialog box
List

```
WINDOW 1, "Test", (25,40) - (125,140), 1
  PRINT "Type 1"
WINDOW 2, , (150,40) - (250,140), 2
  PRINT "Type 2"
WINDOW 3, , (275,40) - (375,140), 3
  PRINT "Type 3"
WINDOW 4, , (400,40) - (500,140), 4
  PRINT "Type 4"
```

FIGURE 14-1

The four types of windows created by BASIC

Type 1: The document window

The default Output window is really a document window. These windows have a *title bar* located at the top of the window, which contains the window's title, the horizontal lines that appear when the window is selected, and the *close box* in the upper left corner. They can also contain a *size box* in the lower right corner, used to drag the window larger or smaller, as we've often done ourselves. This type of window can be moved on the screen by dragging the title bar.

Document windows are also provided with empty, vertical *scroll bars* along the right side of the window. Creating windows that scroll is an advanced programming technique that's beyond the scope of this book. However, the scroll bar does affect some aspects of our program output, and we'll discuss these as they occur.

Type 2: The bordered dialog box

The familiar Save dialog box that we've used many times is an example of the bordered dialog box. The *alert box* that appears when an error has occurred is also a bordered dialog box. Unlike the document window, this type of window cannot be moved or resized.

Type 3: The plain dialog box

The plain dialog box is just like the bordered dialog box, except that its border is only one thin line.

Type 4: The shadowed dialog box

The shadowed dialog box is similar to both the bordered and plain dialog boxes, except that it has a shadow behind it for visual effect.

THE WINDOW STATEMENT

You have probably figured out that each of the four different windows in Figure 4-1 was created by a single WINDOW statement. Let's take a closer look at this powerful statement, and discuss its parameters. The general syntax for the WINDOW statement is:

WINDOW *ID, Title, Rectangle, Type*

Now let's look at each of the parameters individually and see how they let us create windows.

The ID parameter

Since we have so much flexibility in the number and type of windows we can use in our programs, BASIC must have some way of keeping them straight. This is the purpose of the *ID* parameter. We can have up to four different windows defined in a BASIC program, and they are given *ID* numbers 1 through 4.

The default Output window is automatically assigned window *ID* number 1. So if we want to continue using the default Output window, any other windows our program creates must not have *ID* number 1.

The Title parameter

The *Title* parameter of the WINDOW statement only applies to document windows, which have a title bar. Notice in our example in Figure 14-1 that we've assigned the title *Test* to the first window we created.

You might be wondering how this affects our *program* titles. You will recall that when we choose Save As... from the File menu and save a program, the name we give it materializes in the title bar of the Output window. If we later use a WINDOW statement to create a document window with *ID* number 1, any title we include in the WINDOW statement will override the program's title and appear in the title bar. If we then save the program it will *not* be saved under the name that we created inside the WINDOW statement, and the title will revert to the original name. However, if we leave the default Output window alone and create other document windows with *ID* numbers 2, 3, or 4, the default Output window will continue to display the program title and the other windows will display the names we've assigned to them.

None of the three other types of windows uses the *Title* parameter, so if you include a title, it will be ignored. Therefore, the title string can be omitted as long as a comma is there to mark the place of the missing parameter. Notice that the WINDOW statements in Figure 4-1 that create windows 2, 3, and 4 do not have a *Title* parameter.

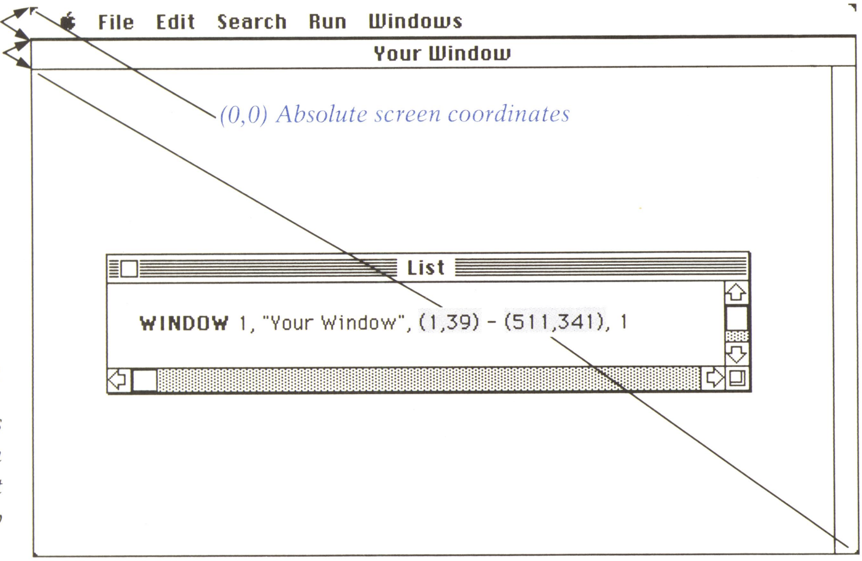
The Rectangle parameter

The *Rectangle* parameter specifies the location of the window on the screen. It is expressed as two pairs of (x,y) coordinates: The first pair defines the upper left corner of the window, and the second pair defines the lower right corner.

As we will discuss shortly, the *Rectangle* coordinates are *absolute* screen coordinates, referenced from the very upper left corner of the *screen*, which has the (x,y) coordinate (0,0). The (x,y) coordinates that we've been using for things like PSET, LINE, and CIRCLE are *relative* window coordinates, which are referenced from the upper left corner of the *Output window.*

Figure 14-2 shows a WINDOW statement that creates a document window that completely fills the screen.

Notice that, when specifying the upper left corner of a document window, we must take into account the 20-pixel space that will be occupied by the *menu bar* and another 19 pixels for the *title bar* of the window itself.



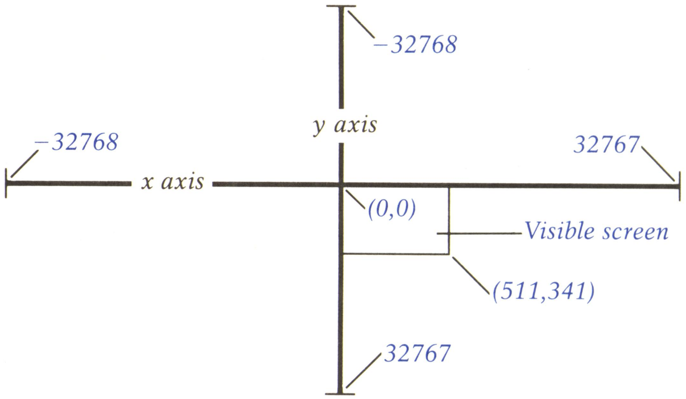
```
WINDOW 1, "Your Window", (1,39) - (511,341), 1
```

FIGURE 14-2

The coordinates for the maximum visible document window

FIGURE 14-3

The maximum visible screen and offscreen-window coordinates



As a result, the actual upper left corner of the window is located 39 pixels below the top of the screen. That's why we see the coordinate (1,39) instead of (1,1).

The title bar consideration does not apply to the other three types of windows, which do not have title bars. However, the bordered dialog box (window type 2) does have a border that extends 10 pixels beyond the boundaries of the specified position rectangle.

The window coordinates in our example, (1,39)–(511,341), specify the coordinates of the largest *visible* window. However, windows can *actually* be much larger. For example, when we drag the Output window so that part of it is offscreen, the Macintosh knows that it's still there, which is why we can drag it back onto the screen. And because the Macintosh can keep track of offscreen coordinates, we can put the borders of a window *outside* the screen. For instance, if a window starts at (−1,−1) and ends at (512,342), *every* pixel on the screen (other than the menu bar) will be *inside* the window even though part of the window won't be showing on the screen. As shown in Figure 14-3, the maximum coordinates for a window larger than the visible screen are −32768 to 32767.

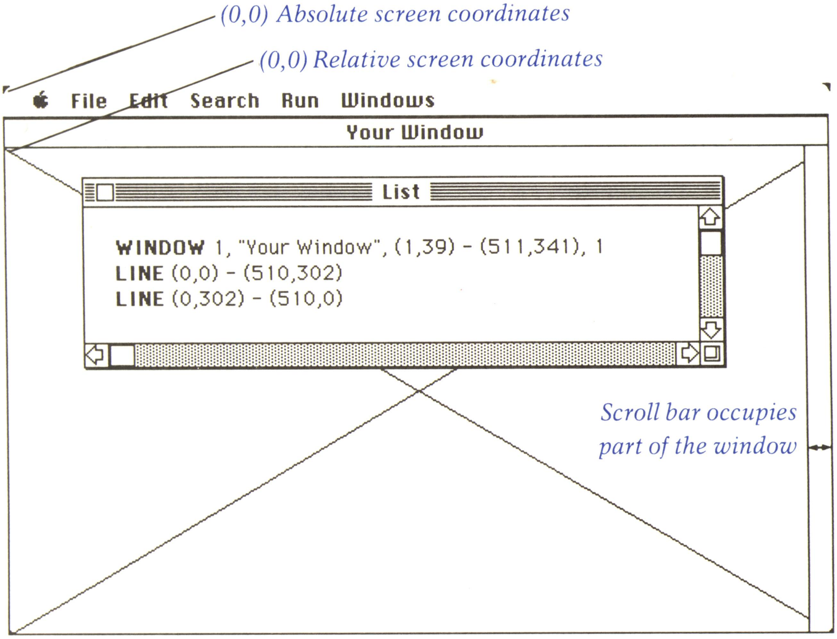
The Type parameter

We've seen four different types of windows we can create in our BASIC programs. The *Type* parameter specifies the type of window we want to create. If no *Type* parameter is specified, BASIC defaults to the document window (type 1). Here is a list of the window types preceded by their parameter numbers:

1—Document window with title bar, size box, and close box
2—Bordered dialog box
3—Plain dialog box
4—Shadowed dialog box

ABSOLUTE VERSUS RELATIVE COORDINATES

We mentioned earlier that when we specify the position of a window, we do so using absolute screen coordinates. Once a window has been created, however, all the activity that occurs within the window is defined in terms of the *relative coordinates* inside it. This is illustrated in Figure 14-4.



(0,0) Absolute screen coordinates

(0,0) Relative screen coordinates

```
WINDOW 1, "Your Window", (1,39) - (511,341), 1
LINE (0,0) - (510,302)
LINE (0,302) - (510,0)
```

Scroll bar occupies part of the window

FIGURE 14-4

Absolute and relative coordinates

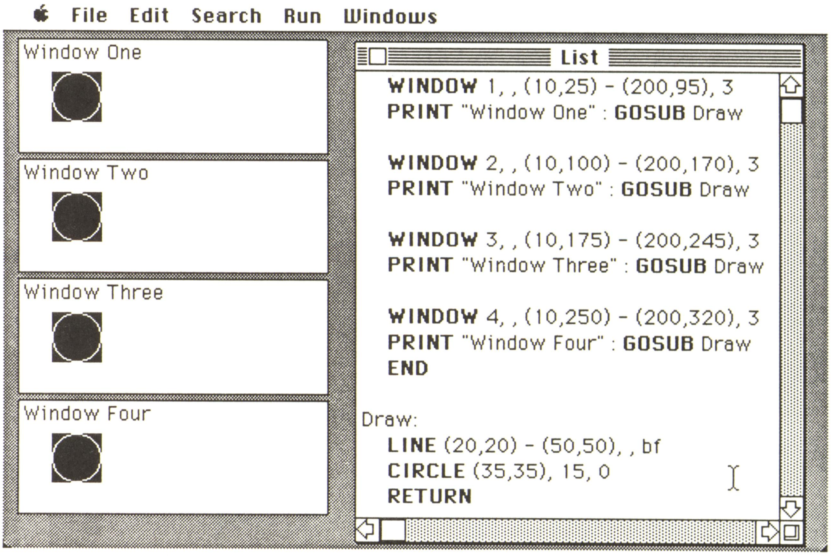
Having created a document window, we've added two LINE statements to draw lines between the opposite corners. Notice that the first line begins at the upper left corner of the window, having relative window coordinates of (0,0). Also notice that the right ends of both lines are hidden by the scroll bar on the right side of the window.

So, even though we've created the largest visible document window with the coordinates (1,39) – (511,341), we see that only a portion of the total screen area is available for use by the window. We'll see more examples of how we can use relative window coordinates to position text and graphics inside a window.

CREATING MULTIPLE WINDOWS

Our first window example in Figure 14-1 illustrated that we can create more than one window with a program. Let's explore this idea further, and learn more about absolute and relative coordinates.

The example program shown in Figure 14-5 creates four "plain" (type-3) windows on the screen. As each window is created, a PRINT statement prints an identifying label. Then, the GOSUB statement directs the program to the subroutine labeled *Draw,* which displays some graphics in the window. This illustrates how each window is like an independent output screen. Output in Window Three, for example, has no effect on Windows One, Two, and Four.



```
    WINDOW 1, , (10,25) - (200,95), 3
    PRINT "Window One" : GOSUB Draw

    WINDOW 2, , (10,100) - (200,170), 3
    PRINT "Window Two" : GOSUB Draw

    WINDOW 3, , (10,175) - (200,245), 3
    PRINT "Window Three" : GOSUB Draw

    WINDOW 4, , (10,250) - (200,320), 3
    PRINT "Window Four" : GOSUB Draw
    END

Draw:
    LINE (20,20) - (50,50), , bf
    CIRCLE (35,35), 15, 0
    RETURN
```

FIGURE 14-5

Four type-3 windows created by one program

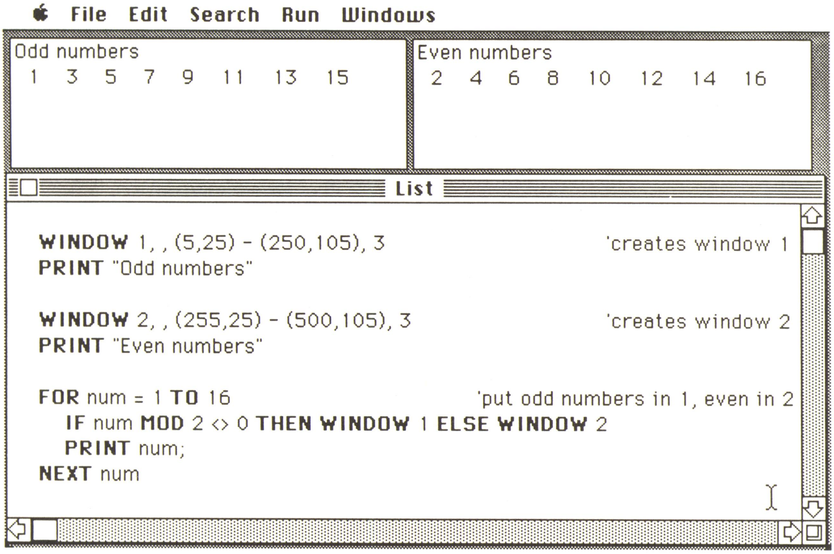
Notice that the graphics displayed in each window are in the same *relative* position in all four windows. This demonstrates the effect of the relative coordinates that are used to specify the position of output in a window. Each window has its own upper left corner, which has the relative coordinate (0,0). When we specify the position of text or graphics in a window, it is always referenced from the upper left corner of the window.

It's important to realize that when BASIC statements create either text or graphics output, it is directed to the window that was last referenced by a WINDOW statement. In our example program, both the text and graphics output are performed immediately after the window has been created, and before the next WINDOW statement is encountered by the program. If our program were to perform additional output after the *WINDOW 4* statement, it would all appear in Window Four. Let's see how we can continue to use multiple windows after they have been created.

Output to multiple windows

Although our BASIC programs can create up to four windows simultaneously, only one window at a time can receive output from the program. We must specify which window is to be the *active* one with a WINDOW statement having only the *ID* parameter.

The example program shown in Figure 14-6 creates two windows and then outputs odd numbers to one window and even numbers to the other. This two-window output is accomplished with a FOR...NEXT loop



```
Odd numbers
 1   3   5   7   9   11   13   15
```

```
Even numbers
 2   4   6   8   10   12   14   16
```

```
WINDOW 1, , (5,25) - (250,105), 3                         'creates window 1
PRINT "Odd numbers"

WINDOW 2, , (255,25) - (500,105), 3                       'creates window 2
PRINT "Even numbers"

FOR num = 1 TO 16                          'put odd numbers in 1, even in 2
  IF num MOD 2 <> 0 THEN WINDOW 1 ELSE WINDOW 2
  PRINT num;
NEXT num
```

FIGURE 14-6

Output to two windows

that generates the numbers 1 through 16. We're using the MOD operator to determine if each number is evenly divisible by 2 (a sure test for an even number). The MOD operator in this case returns a value of 0 if the number is even, and 1 if the number is odd.

The values returned by the MOD operator are used in the IF... THEN...ELSE statement to activate one window or the other using the simplified WINDOW statement. Once it is activated, this window receives the output generated by the PRINT statement that displays the number.

In this section, we've learned how to create the four different types of window that we can use in our programs. We've seen how absolute screen coordinates specify the position of the window, and how relative window coordinates specify the position of the window's contents. In the next section, we'll learn the essential technique of using several document windows in our programs.

USING DOCUMENT WINDOWS

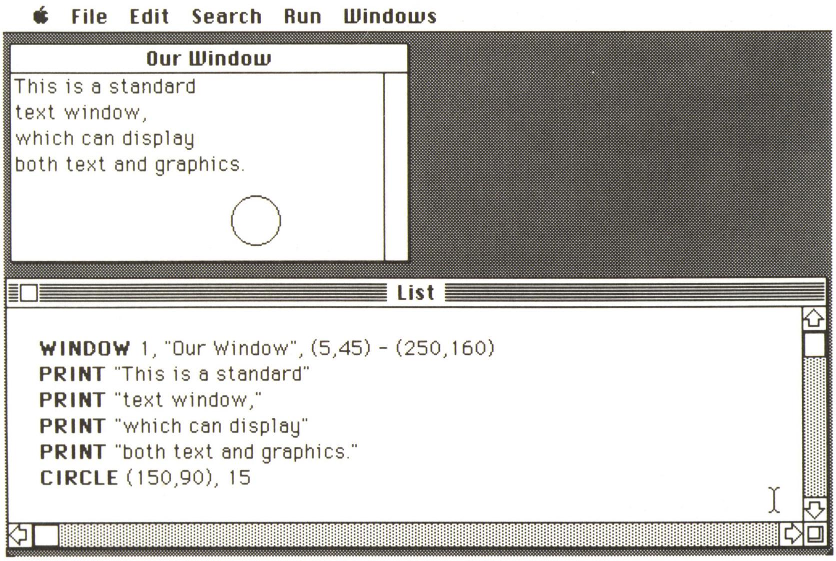
Of the four different types of window that we can create, the document window is the most flexible. That's because we can reposition it on the screen by dragging its title bar with the mouse, and also change its size and close it. Let's experiment with document windows, and in the process discover a major consideration when using this type of window: *refreshing*.

Document windows are flexible

While all four window types can be used to display text and graphics, the document window is especially suited for this purpose. Let's use a simple example program to demonstrate this.

As shown in Figure 14-7, we can easily create a document window and then output both text and graphics. Notice that the circle we've drawn has its center located at relative window coordinates (150,90). Let's see what happens when we use the mouse to drag the window to a new position on the screen, and enlarge it by dragging the size box.

As we can see in Figure 14-8, the window's position and size have been changed, yet the text and graphics in the window appear unchanged. Both are still displayed at the same relative position within the window. The center of the circle is still located at relative coordinates (150,190).

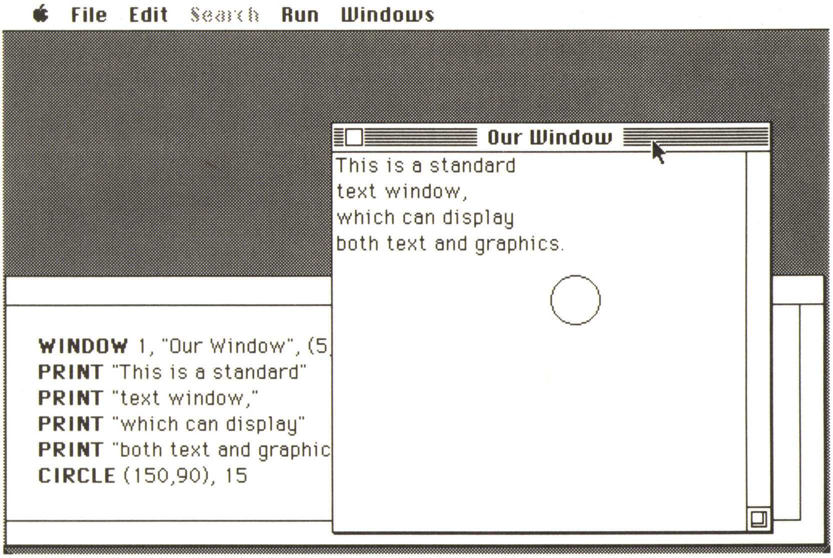

File
Edit
Search
Run
Windows
Our Window
List

This is a standard
text window,
which can display
both text and graphics.

```
WINDOW 1, "Our Window", (5,45) - (250,160)
PRINT "This is a standard"
PRINT "text window,"
PRINT "which can display"
PRINT "both text and graphics."
CIRCLE (150,90), 15
```

FIGURE 14-7

A document window with text and graphics


File
Edit
Search
Run
Windows
Our Window

This is a standard
text window,
which can display
both text and graphics.

```
WINDOW 1, "Our Window", (5,
PRINT "This is a standard"
PRINT "text window,"
PRINT "which can display"
PRINT "both text and graphic
CIRCLE (150,90), 15
```

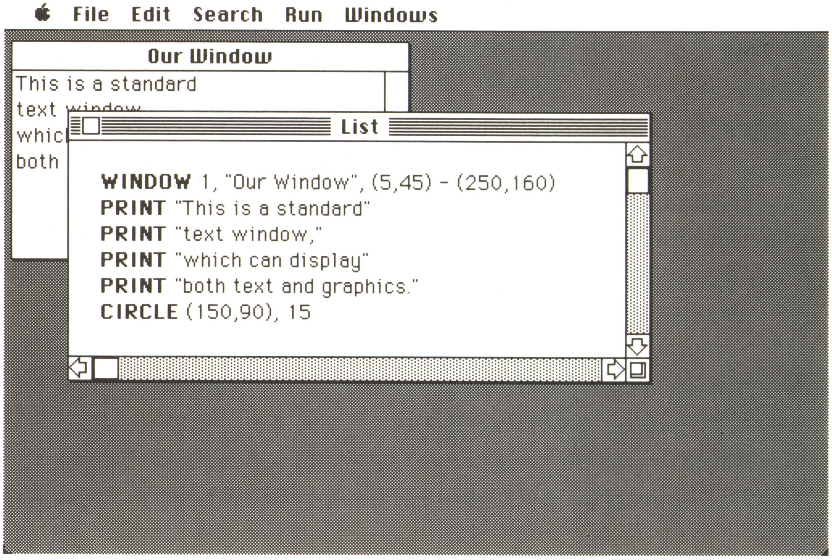
FIGURE 14-8

The window's contents move with it

But our document window isn't the only window that can be moved around the screen like this. We can also move the program List window in the same manner. For example, after running this same program, shown in Figure 14-9, we moved the List window to partially cover our Output window. While we can see that some of the contents of the document window are still there, other parts seem to be hidden underneath the List window.

Let's see if everything is still in the document window. We can reactivate it simply by clicking in it. As shown in Figure 14-10, when we do this the document window again comes to the front but part of its contents have been wiped out by the List window. This happens because even though we are using two *windows,* there is only one *screen* and everything must happen there. When we moved the List window on top of our program's window, all the pixels in the overlapped area were changed to display the upper left corner of the List window.

This is a real concern to us because we would like to be able to fully utilize the features of the document window, like moving it and changing its size. If we are going to allow other windows to temporarily cover our document window, then we'll have to find some method of *restoring* the contents. In Macintosh parlance, this is called *refreshing a window.*

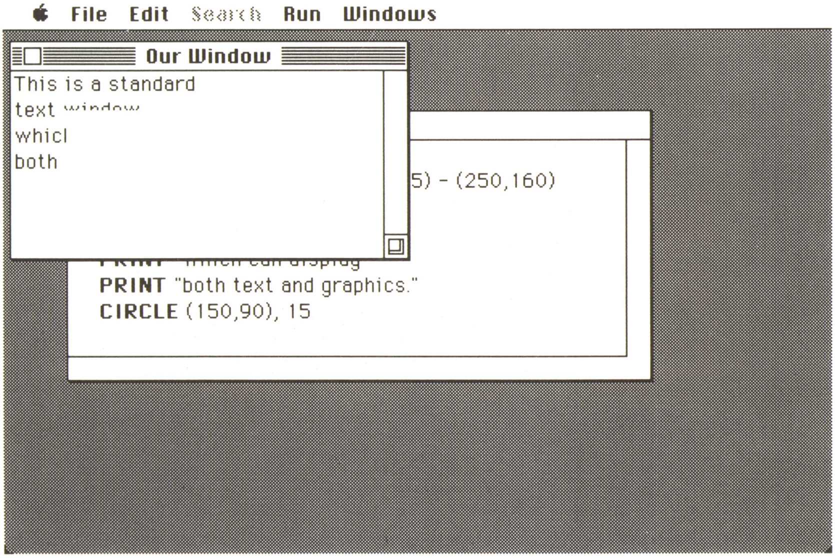


```
This is a standard
text
whic
both
```

```
WINDOW 1, "Our Window", (5,45) - (250,160)
PRINT "This is a standard"
PRINT "text window,"
PRINT "which can display"
PRINT "both text and graphics."
CIRCLE (150,90), 15
```

FIGURE 14-9

Overlapping windows can be hazardous



```
This is a standard
text window
whicl
both
```

```
5) - (250,160)
PRINT "both text and graphics."
CIRCLE (150,90), 15
```

FIGURE 14-10

Contents of covered window are lost

Refreshing a window

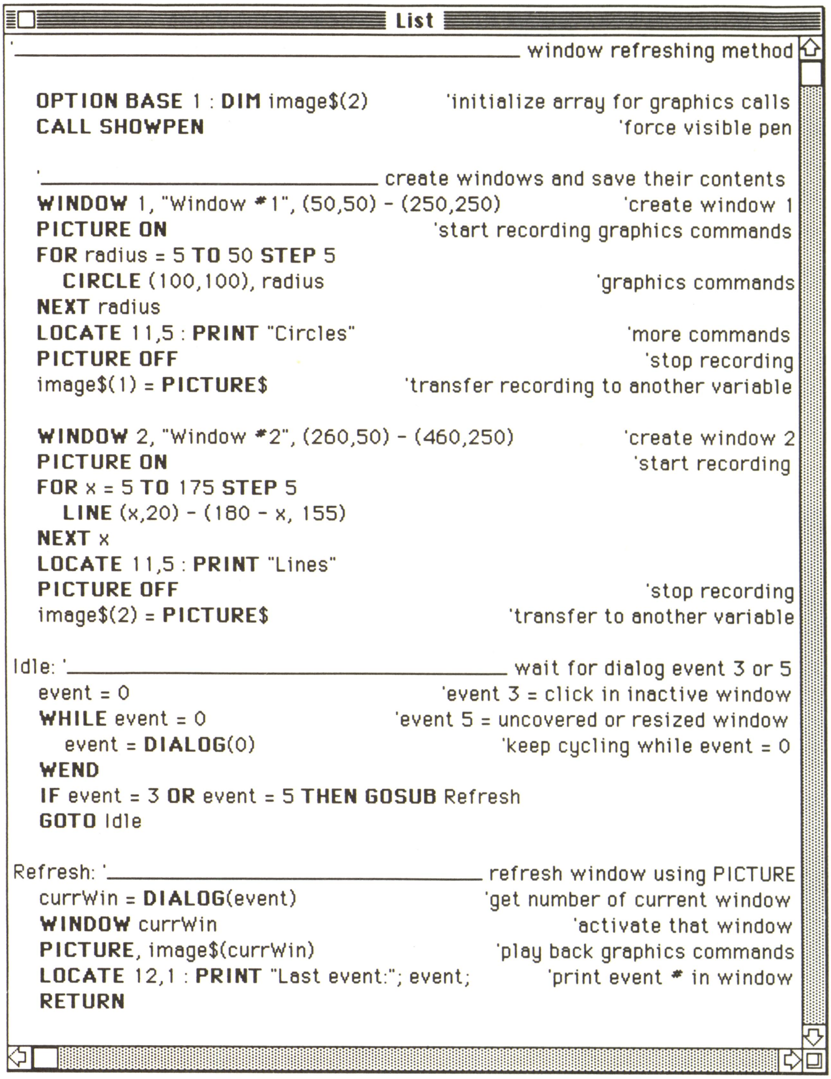
The concept of refreshing a window is fairly complex, so you shouldn't worry if it isn't crystal clear after reading through the next section only once. Take your time, and examine our example programs thoroughly. Although it is complex, the process is very logical and can be implemented in your own programs.

The key to refreshing a window is to *save a recording* of all the graphic and text activity that takes place in the window. Then, when the need to refresh the window arises, we can use this recording to regenerate the window's contents.

There are four aspects to the window-refreshing process, each employing a BASIC statement or function. These appear in the following order in a program:

1. PICTURE ON and PICTURE OFF statements: Capture all output to the window.
2. PICTURE$ function: Save the output for future use.
3. DIALOG function: Determine the need to refresh the window.
4. PICTURE statement: Refresh the window.

The program in Figure 14-11 demonstrates each of these steps. The program creates two document windows, performs text and graphic output to both windows, and refreshes a window if necessary. Window #1 contains several concentric circles, and Window #2 contains several lines drawn to form a graphic design. We said earlier that the first step in the window-refreshing process is to capture all the program's output operations. Let's take a look at the program to see how this is accomplished.



```
'______________________________________________________ window refreshing method

  OPTION BASE 1 : DIM image$(2)          'initialize array for graphics calls
  CALL SHOWPEN                                             'force visible pen

  '_______________________________________ create windows and save their contents
  WINDOW 1, "Window #1", (50,50) - (250,250)                'create window 1
  PICTURE ON                                 'start recording graphics commands
  FOR radius = 5 TO 50 STEP 5
    CIRCLE (100,100), radius                                 'graphics commands
  NEXT radius
  LOCATE 11,5 : PRINT "Circles"                               'more commands
  PICTURE OFF                                                 'stop recording
  image$(1) = PICTURE$                   'transfer recording to another variable

  WINDOW 2, "Window #2", (260,50) - (460,250)                 'create window 2
  PICTURE ON                                                  'start recording
  FOR x = 5 TO 175 STEP 5
    LINE (x,20) - (180 - x, 155)
  NEXT x
  LOCATE 11,5 : PRINT "Lines"
  PICTURE OFF                                                 'stop recording
  image$(2) = PICTURE$                            'transfer to another variable

Idle: '_______________________________________________ wait for dialog event 3 or 5
  event = 0                                   'event 3 = click in inactive window
  WHILE event = 0                         'event 5 = uncovered or resized window
    event = DIALOG(0)                             'keep cycling while event = 0
  WEND
  IF event = 3 OR event = 5 THEN GOSUB Refresh
  GOTO Idle

Refresh: '_______________________________________ refresh window using PICTURE
  currWin = DIALOG(event)                          'get number of current window
  WINDOW currWin                                       'activate that window
  PICTURE, image$(currWin)                         'play back graphics commands
  LOCATE 12,1 : PRINT "Last event:"; event;             'print event # in window
  RETURN
```

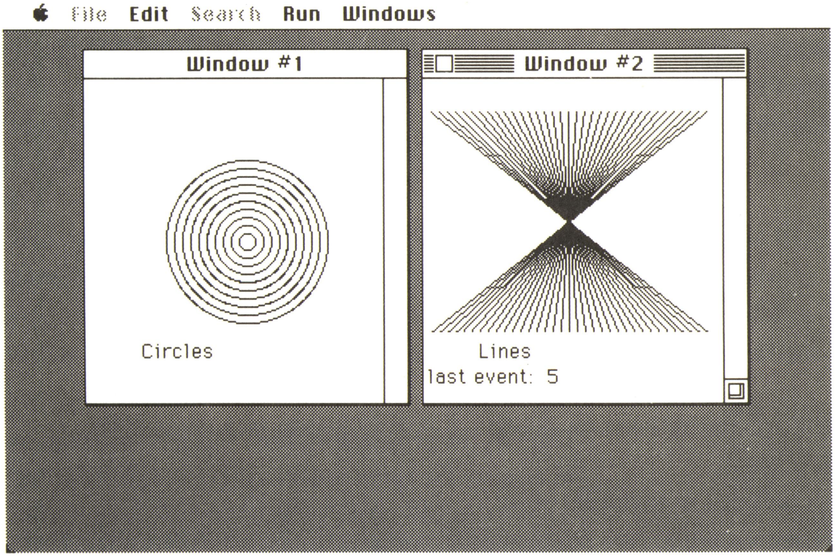
FIGURE 14-11

A program that creates and refreshes two document windows containing text and graphics

(continued)

FIGURE 14-11

(continued)



last event: 5

Capturing window output with PICTURE ON and PICTURE OFF

We'll skip the first section of the program for now and begin our discussion with the section that creates the windows. After Window #1 is created, the next statement is:

```
PICTURE ON
```

This statement instructs BASIC to capture any window output that follows so that it can be used later to refresh the window if necessary. The next few statements actually produce the screen output to be captured in Window #1. Then we see the statement:

```
PICTURE OFF
```

This tells BASIC to stop capturing output.

We might think of these two statements as turning on and turning off a "tape recorder" inside BASIC. PICTURE ON starts a recording that captures the following graphic and print commands as they're output to the window, and PICTURE OFF stops the recording. Next, we need to save this section of the "tape" with a unique name, so we can have BASIC "play it back" for us when we need it. The PICTURE$ function is our tool for this.

Saving the output with the PICTURE$ function

BASIC uses the PICTURE ON statement to store each window-output operation as a special code in the PICTURE$ function. In practice, PICTURE$ acts like a string variable, which is why it contains a dollar sign. But since the reserved-word PICTURE$ is needed to capture the output to *each* window created by a program, we must reassign its contents to a second string variable with a name that's unique to each window. In effect, this "empties" the PICTURE$ function so that it can be used with another window. This is what we accomplished with the last statement in the section that creates Window #1:

```
image$(1) = PICTURE$
```

Later in the program we use this string to "play back" the window output in exactly the same order as it originally occurred.

At this point we need to look back to the first section of the program. Notice that we've used a DIM statement to initialize a string array named *image$* that can hold two strings (one for each window's output). The statement *image$(1) = PICTURE$* saves the captured output operations in the first element of the string array.

After Window #1 is created and its output saved in a string variable, Window #2 is created. The unique, fan-like output in this window is likewise captured by the PICTURE ON and PICTURE OFF statements, and is then saved in the second element of the string array with the name *image$(2)*. We now have the output for both windows saved in a string array that we will use to refresh each window when the need arises.

Determining the need to refresh with the DIALOG function

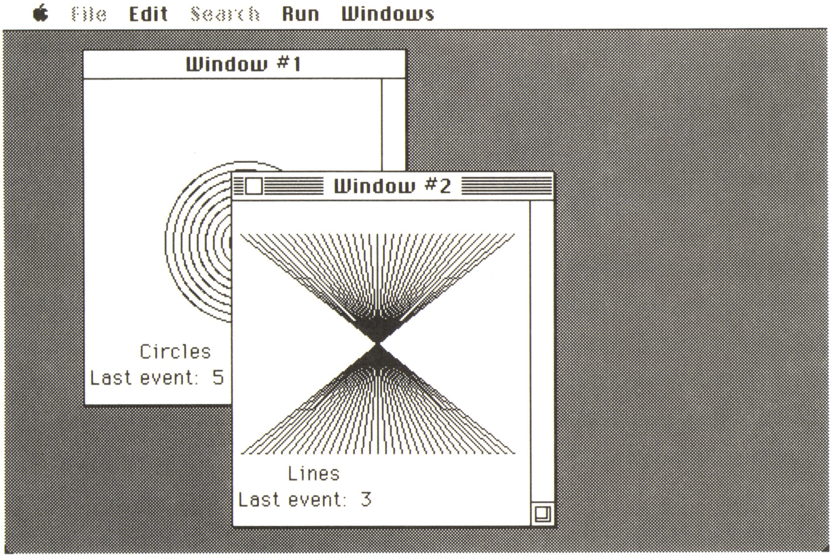
When the program enters the *Idle* section, it loops, waiting for a dialog event to occur that tells BASIC that a window needs to be refreshed. There are two reasons that a window may need to be refreshed: The user clicked the mouse in an inactive window, bringing it to the front of the screen and making it active, or part of a covered window has been uncovered, either by resizing it, by moving a window that covered it, or by closing a window that covered it. These two conditions have been assigned specific event numbers that the DIALOG function can respond with:

3—An inactive window has been clicked
5—A window has been resized or uncovered

In our program, an IF...THEN statement in the *Idle* section first determines if one of these two events has occurred, and if so, it sends the program to the subroutine *Refresh* (discussed shortly) which actually refreshes the window. If the dialog event is not 3 or 5, the program goes back to the idle loop.

Let's look at an example event 3, clicking an inactive window to make it active.

As shown in Figure 14-12, we've dragged Window #2 so that it covers part of Window #1. This action has wiped out part of the contents of Window #1, so it will have to be refreshed if the user wants to see it again. The simplest way to see all of a window that is partially covered is to click the mouse in it. This is what we've done in Figure 14-13 on the following page.

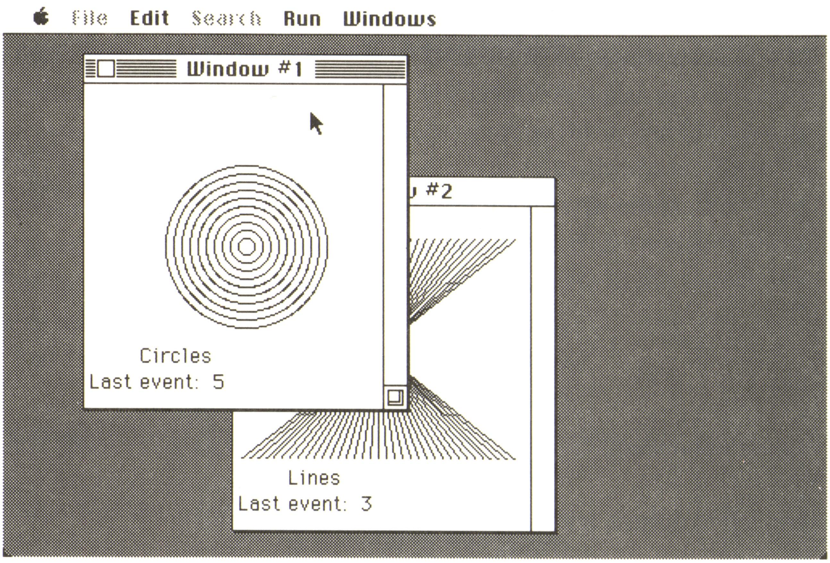


```
Circles
Last event:  5
```

```
Lines
Last event:  3
```

FIGURE 14-12

Window #2 covering Window #1



```
Circles
Last event: 5
```

```
Lines
Last event: 3
```

FIGURE 14-13

Clicking Window #1 to make it active and refresh its contents

We can see in the figure that Window #1 is now the front window and all of its contents have been refreshed by the program! The two windows are still in the same position on the screen, however, so now Window #2 is partially covered.

Refreshing the window with the PICTURE statement

Once we've used the DIALOG function to determine that a window needs to be refreshed, we must find out *which* window to refresh. This is accomplished by a second DIALOG function, which occurs in the first statement of the *Refresh* subroutine of the program. The statement:

```
currWin = DIALOG(event)
```

returns the window *ID* number (1 through 4) of the window that was affected by the dialog event which occurred, regardless of whether it was event 3 or 5. In our program, we'll expect the window *ID* to be either 1 or 2, since we're using only those two windows.

Next, we use a simple WINDOW statement containing only the window *ID* to activate the window needing refreshment. The statement:

```
PICTURE, image$(currWin)
```

is the one that actually does the window refreshing. It uses the information that was saved earlier in the string variable *image$* to completely recreate all the output that was originally performed in the window. Because *image$* is an array, we're using the window *ID* number as the index to the array. This ensures that we'll refresh only the window that needs it.

ADVANCED WINDOW-MANAGEMENT TECHNIQUES

Working through our current program has given us a good understanding of the basic principles for creating and refreshing windows. But as you've probably surmised, programming with windows is a complex process, and there's more to learn before we leave the subject. The knowledge we've gained so far will serve us in good stead as we explore two advanced issues concerning windows: queuing dialog events and active-window management techniques.

Queuing dialog events

Let's do an interesting experiment that illustrates just how BASIC and the Macintosh handle events. First, run the current window program again so that the windows return to their original, non-overlapping positions and then click once in each window to generate an event 3 for each. Next, choose Note Pad from the Apple menu; when it appears, it partially obscures both windows. Now, click the Note Pad's close box. Notice that *both* windows are refreshed, since an event 5 was generated for each, as indicated in the *Last event* line in the window. In this case, one action (closing the Note Pad window) caused two events (refreshing both windows). This points out that BASIC has the ability to save up, or *queue,* a series of events and respond to each in turn.

Let's run our program again, and drag Window #2 so that it partially covers Window #1. Now let's click in Window #1 and watch the *Last event* line in the window: Notice that first an event 3 occurs (making the inactive window active and refreshing it), followed immediately by an event 5 (since part of the covered area has been revealed), which refreshes the window *again.* Once again, BASIC *queued* the dialog events and responded to each in turn.

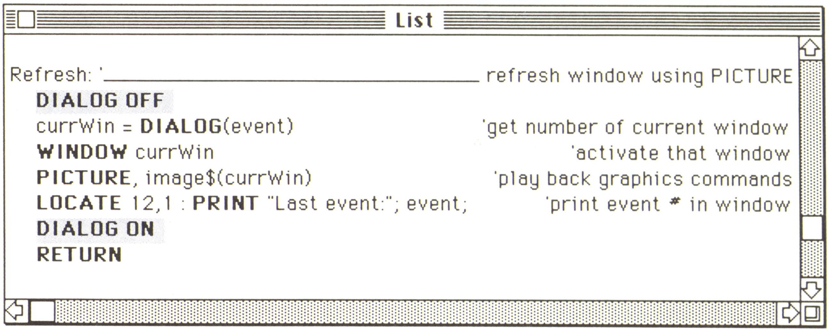
It might have occurred to you that it's a bit redundant to have event 5 refresh the window a second time, just after it's been nicely refreshed by event 3. That's true. And although it's not much of a problem in this simple program, it can be in a program that draws complex graphics, which can take a while to be redrawn each time. We can prevent BASIC from queuing up a second, unnecessary event with the DIALOG OFF and DIALOG ON statements.

The DIALOG OFF and DIALOG ON statements

DIALOG OFF *disables* event trapping, and DIALOG ON *enables* it again. We've seen in our current program that event trapping is always on, which is why events 3 and 5 "stack up" in a queue and are both processed even if only one of them is necessary. What we want our program to do is get an event, turn off event trapping while that event is processed, then turn event trapping back on so it can respond to another event.

Since we want our program to deal only with an event 3 or an event 5 (whichever occurs), we need to put a DIALOG OFF statement at the beginning of the *Refresh* subroutine, and a DIALOG ON at the end, just before the RETURN that sends the program back to the idle loop. Since event trapping is turned back on again by DIALOG ON, the *Idle* routine is ready to receive another event. Figure 14-14 displays the *Refresh* subroutine with the DIALOG OFF and DIALOG ON statements added.

Now when we run our program and click on a covered window, it pops to the front, becomes active, and is refreshed only once. The second, redundant event 5 is ignored because it is prevented from entering the queue. These two simple statements make our program run much more efficiently. They are also critical for the next stage of our program: active-window management.



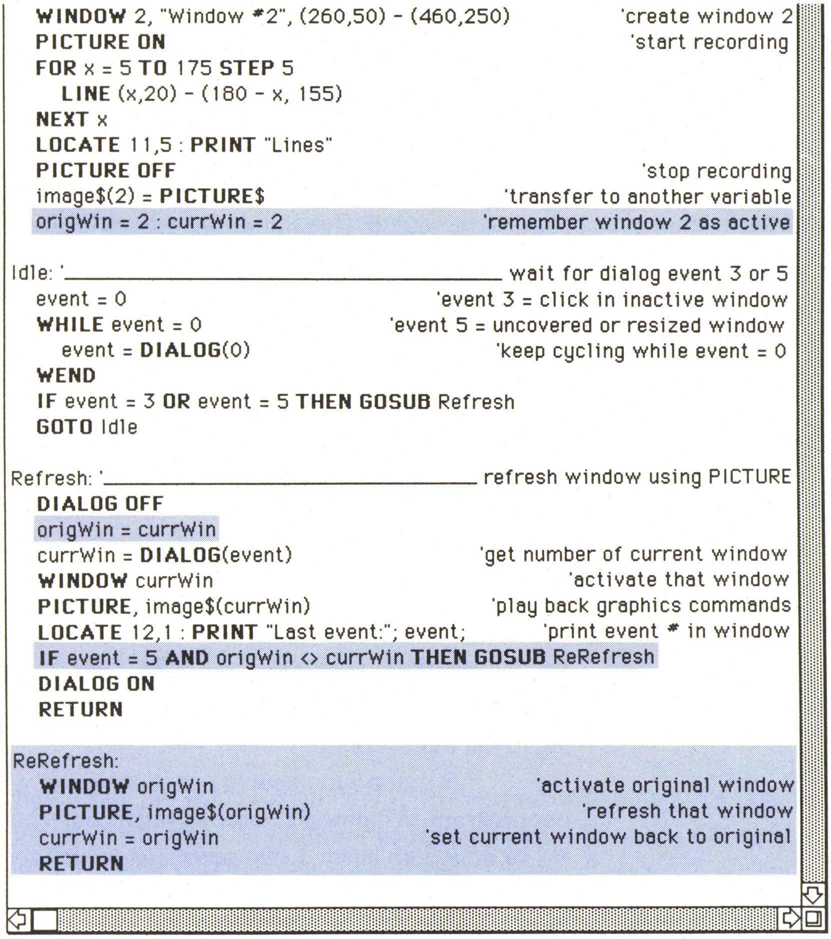
```
Refresh: '_________________________________ refresh window using PICTURE
  DIALOG OFF
  currWin = DIALOG(event)                        'get number of current window
  WINDOW currWin                                      'activate that window
  PICTURE, image$(currWin)                      'play back graphics commands
  LOCATE 12,1 : PRINT "Last event:"; event;        'print event # in window
  DIALOG ON
  RETURN
```

FIGURE 14-14

Using DIALOG OFF and DIALOG ON so that our program processes one event at a time

ACTIVE-WINDOW MANAGEMENT

During your programming sessions, there have probably been times when you've covered up the List window with the Output window. When you dragged the Output window and uncovered the List window, you might have noticed that the Output window *remained active* even as the List window was refreshed. This makes sense: You would like the window you move to stay active while windows it overlaps are automatically refreshed. So far that doesn't happen with the windows created by our current program. Instead, the covered, inactive window is made active so that it can be refreshed, and then it *stays active.* How can we both refresh the covered window, and leave the current window active? Figure 14-15 shows the necessary revisions we've made to the second half of the program to accomplish this.



```
  WINDOW 2, "Window #2", (260,50) - (460,250)         'create window 2
  PICTURE ON                                           'start recording
  FOR x = 5 TO 175 STEP 5
    LINE (x,20) - (180 - x, 155)
  NEXT x
  LOCATE 11,5 : PRINT "Lines"
  PICTURE OFF                                           'stop recording
  image$(2) = PICTURE$                        'transfer to another variable
  origWin = 2 : currWin = 2                   'remember window 2 as active

Idle: '________________________________________ wait for dialog event 3 or 5
  event = 0                                  'event 3 = click in inactive window
  WHILE event = 0                           'event 5 = uncovered or resized window
    event = DIALOG(0)                             'keep cycling while event = 0
  WEND
  IF event = 3 OR event = 5 THEN GOSUB Refresh
  GOTO Idle

Refresh: '________________________________ refresh window using PICTURE
  DIALOG OFF
  origWin = currWin
  currWin = DIALOG(event)                         'get number of current window
  WINDOW currWin                                       'activate that window
  PICTURE, image$(currWin)                        'play back graphics commands
  LOCATE 12,1 : PRINT "Last event:"; event;            'print event # in window
  IF event = 5 AND origWin <> currWin THEN GOSUB ReRefresh
  DIALOG ON
  RETURN

ReRefresh:
  WINDOW origWin                                     'activate original window
  PICTURE, image$(origWin)                                'refresh that window
  currWin = origWin                          'set current window back to original
  RETURN
```

FIGURE 14-15

The program modified to keep the original window active

The first thing we have to do is tell our program to keep track of the original active window. We do this with the variable *origWin.* Since Window #2 is the last window created by the program, we assign its value to *origWin* in the last statement in the section that creates Window #2. We also assign Window #2's value to the variable *currWin* which, recall, is used later in the *Refresh* subroutine to record the number of the window where the event occurred. So when the program is first run, Window #2 is active and its value assigned to both variables.

Let's try this experiment. Drag Window #1 over Window #2, then partway off again. Although this is two events—a 3 when Window #1 is clicked and a 5 when Window #2 is partially uncovered—the *first* event that occurs is an event 3, which is all that BASIC sees the first time through the idle loop. Since *event* is equal to 3, the IF...THEN statement in the *Idle* section sends the program to the subroutine *Refresh.* There, the DIALOG OFF statement disables further event trapping, and BASIC drops to a new statement we've added:

```
origWin = currWin
```

BASIC looks at this statement and sets *origWin* to the number of the active window at the time of the event, which is the current window at that moment. But then BASIC drops to the next line:

```
currWin = DIALOG(event)
```

where the *DIALOG(event)* value tells it that the event happened in window #1. So the program changes the variable *currWin* to 1, then activates and refreshes that window, and prints the *Last event* statement as usual. Now BASIC comes upon a new line:

```
IF event = 5 AND origWin <> currWin THEN GOSUB ReRefresh
```

BASIC gets as far as *IF event=5* in this line, says "Nope, an event 3 occurred," then finishes the subroutine by turning DIALOG ON and returning to the *Idle* section.

Let's stop a moment for an overview of what's happened so far in our program. Window #2 began as active, but when we clicked in Window #1 to drag it an event 3 was generated, which sent BASIC to the *Refresh* subroutine, which activated and refreshed Window #1. So at this point, the program has only processed the first event, event 3, and Window #1 is now active.

But now BASIC is back at the *Idle* section. This time through it sees that an event 5 has occurred in Window #2, which we partially uncovered when we dragged Window #1 off it. The IF...THEN statement sends BASIC off to the *Refresh* subroutine again. Since the event happened in Window #2, *DIALOG(event)* changes the variable *currWin* to 2 and *that* window is refreshed. We've seen that to do this, Window #2 must be activated, which in this case causes it to overlap Window #1, obscuring some of *its* contents. Since Window #1 was originally active, we need to make it active again. So what we need the program to do is reactivate Window #1 and *rerefresh* it. This is where the line:

```
IF event = 5 AND origWin <> currWin THEN GOSUB ReRefresh
```

comes into play. BASIC sees that the current *event* is equal to 5. It next asks itself, "Is the value for *origWin* equal to the value for *currWin*?" The answer in this case is "No," because *origWin* was set to 1 the first time through the program, and *currWin* has just been set to 2 in the *Refresh* subroutine. Since *origWin* is not equal to *currWin*, the GOSUB sends BASIC to process the *ReRefresh* subroutine.

Happily, the subroutine *ReRefresh* is pretty straightforward. The first statement:

```
WINDOW origWin
```

activates Window #1 because its value is currently assigned to *origWin*. The next statement:

```
PICTURE, image$(origWin)
```

refreshes Window #1. The next statement:

```
currWin = origWin
```

then resets the value for *currWin* to the current window's value, and the program returns to the subroutine *Refresh* where DIALOG ON restores event trapping and the program is returned to the idle loop.

Although it took some doing, we now have what we want: The uncovered part of Window #2 has been refreshed, and Window #1 has been reactivated and rerefreshed. Now our program acts like a true Macintosh program: We can drag one window off another and have the dragged window remain active. And now that we understand the active-window management process, we can incorporate it into any of our programs that create multiple windows that might overlap.

USING WINDOWS AS DIALOG BOXES

We learned earlier in this chapter that there are four different types of windows we can create. We've seen several examples of the use of the document window in our BASIC programs. Let's explore the next most common window, the *dialog box.* We'll see how to use a dialog box to obtain text information from the user by including *edit fields* created with the EDIT FIELD statement. We'll also learn how to incorporate buttons into our dialog boxes. When we're finished, you'll have a model program that you can change to suit your own needs.

The EDIT FIELD statement

An edit field is a special feature of Microsoft BASIC on the Macintosh that allows the user to enter information from the keyboard. Until now, we've done this using the INPUT statement, which is effective in itself. The edit field, however, offers two distinct advantages:

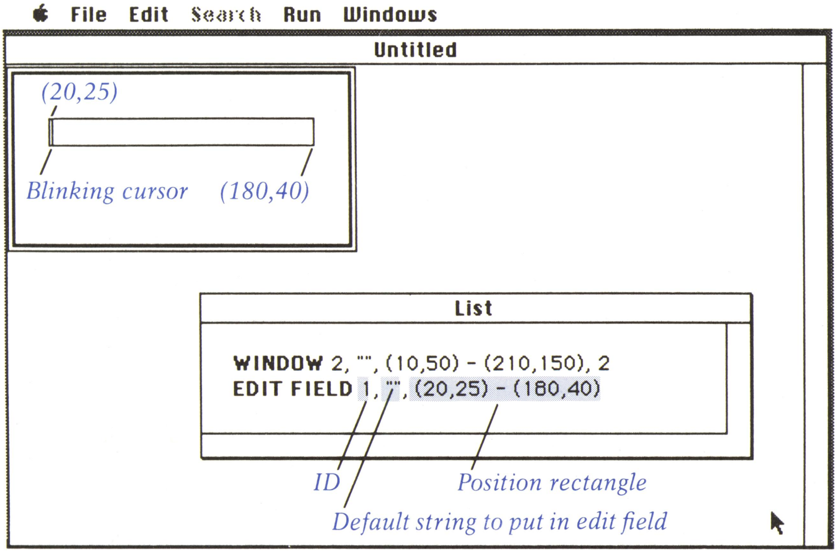
- All standard editing techniques apply to edit fields, including using the mouse to position and select text, and cutting and pasting.
- The user need not press Return to enter data, but can click a button instead.

We'll discuss these differences as we learn how to create and use edit fields. First, let's use a couple of BASIC statements to create a dialog box with one edit field.

The dialog box in Figure 14-16 is created with one WINDOW statement, as Window #2. Notice that we've created a type-2 window: a bordered dialog box. The next statement creates the edit field in that dialog box. Here is a common syntax of the EDIT FIELD statement:

EDIT FIELD *ID, Default String, Rectangle*

We'll discuss each of these parameters in the following sections.


File
Edit
Search
Run
Windows
Untitled
(20,25)
Blinking cursor
(180,40)
List
ID
Position rectangle
Default string to put in edit field

```
WINDOW 2, "", (10,50) - (210,150), 2
EDIT FIELD 1, "", (20,25) - (180,40)
```

FIGURE 14-16

A dialog box with one edit field

The ID parameter

Because we can create more than one edit field in a window, each one is assigned an *ID* number. This is very much like the *ID* numbers that are assigned to each window we create. We'll use this *ID* number to work with the edit field later in our program.

The Default String parameter

When we create an edit field, we can use the *Default String* parameter to tell BASIC to put a particular string of characters inside it, which can be a constant or a variable. We'll use this feature in our next example. Notice in Figure 14-15 that we specified the null string (""), so nothing appears in the edit field.

The Rectangle parameter

We specify the location of the edit field inside the window we've created with the *Rectangle* parameter. Here again, the *Rectangle* is specified in the familiar format: the first (x, y) coordinate is the upper left corner, and the second is the lower right corner. Just remember that these are relative window coordinates, not absolute screen coordinates.

Using edit fields

Now that we have seen how easy it is to create an edit field, let's go on to make some use of it. First, we will learn how to place default information in the edit field and then use standard Macintosh editing techniques to change it. Finally, we'll see how to get the information from an edit field into our program.

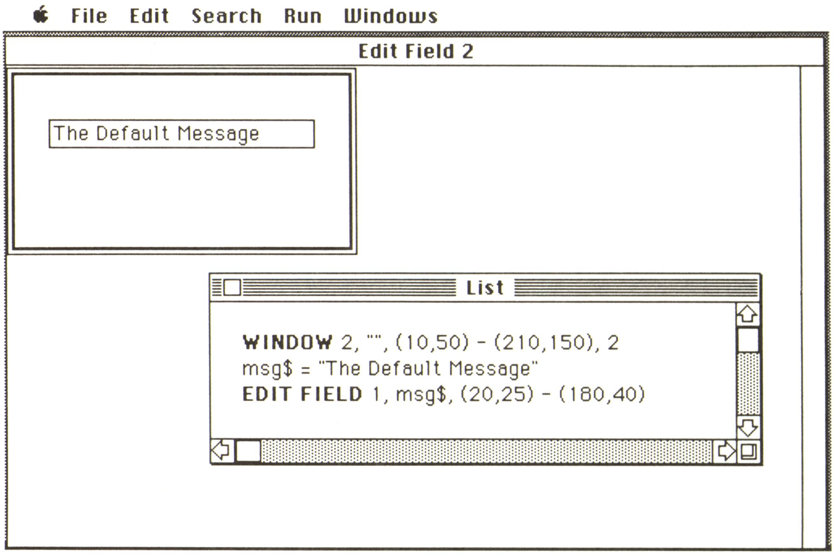
Setting up the Default String

In our last example, we chose not to place any information in the edit field we created. In Figure 14-17, we can see how easy it is to do.

This is a useful feature in situations where the user has to enter some information that we can anticipate. For instance, when we edit a BASIC program and then save it again, BASIC presents the Save dialog box with the program's name already displayed. We can type a new name if we want, or simply click on the Save button to accept the default name.

As we said, all the standard editing techniques can be used in the edit fields that we create. For example, we could drag the pointer through the word *Default* so only it would be selected, and we could type some other word to replace it, or press the Backspace key to delete it.

Also notice in this program that we first assigned the message string to the string variable *msg$,* then used the variable as the *Default String* in the EDIT FIELD statement. We could also assign each *Rectangle* coordinate to a number variable and use it in the *Rectangle* parameter.



The Default Message

```
WINDOW 2, "", (10,50) - (210,150), 2
msg$ = "The Default Message"
EDIT FIELD 1, msg$, (20,25) - (180,40)
```

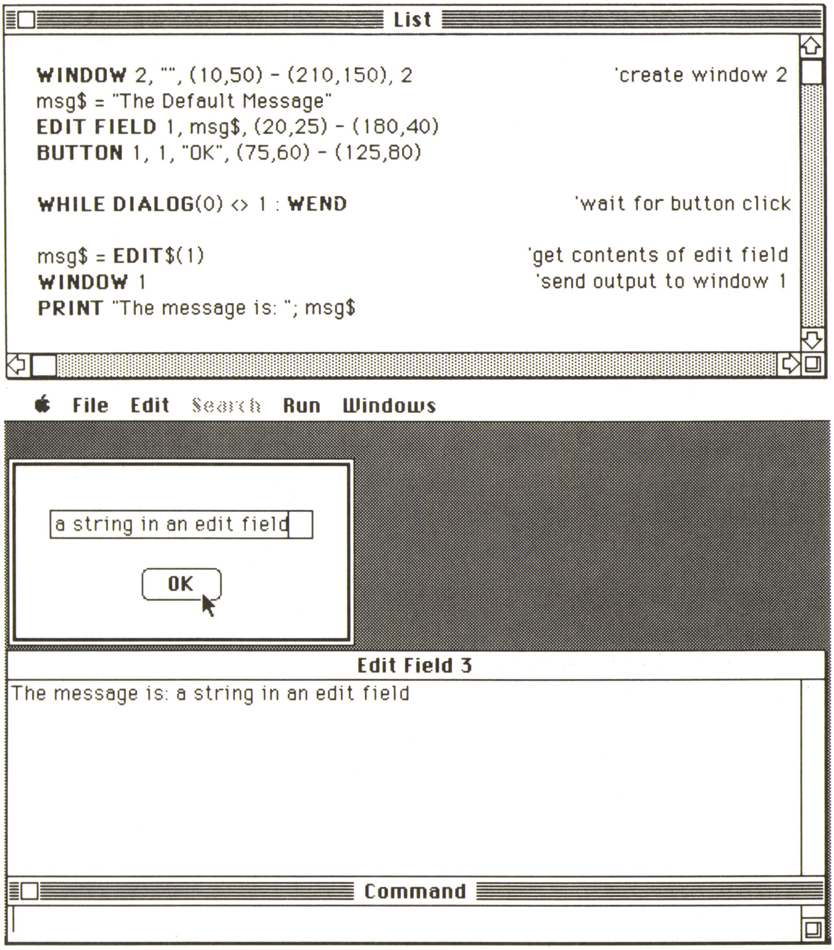
FIGURE 14-17

Displaying a Default String

Using edit field contents with the EDIT$ function

Once we have completed either entering information into an edit field or editing the default contents, we'll need a few more BASIC statements to actually use the information in our program. First, we need to provide the user with some method of telling the program when to accept the contents of the edit field. Old fashioned BASIC would have the user press the Return key, but the typical Macintosh way of doing this is to use a button. Let's see how this is accomplished.

As shown in Figure 14-18, we've added several statements to our program. The first is a BUTTON statement to create one button, labeled OK, in the dialog box. There's nothing fancy about this BUTTON statement; we've created a familiar push button that the user can click.



```
WINDOW 2, "", (10,50) - (210,150), 2                'create window 2
msg$ = "The Default Message"
EDIT FIELD 1, msg$, (20,25) - (180,40)
BUTTON 1, 1, "OK", (75,60) - (125,80)

WHILE DIALOG(0) <> 1 : WEND                        'wait for button click

msg$ = EDIT$(1)                                'get contents of edit field
WINDOW 1                                        'send output to window 1
PRINT "The message is: "; msg$
```

```
The message is: a string in an edit field
```

FIGURE 14-18

Using the contents of an edit field

The WHILE...WEND loop in the middle section of the program is the standard method for detecting a button press that we've used before. When the user clicks the OK button in the dialog box, the DIALOG(0) function returns the value 1, and the program goes ahead and processes the next section.

Here is where the contents of the edit field are accepted for use by the program. This is accomplished with the statement:

```
msg$ = EDIT$(1)
```

The EDIT$ function is what does all the work of acquiring the contents of the edit field for the program. It requires one argument, enclosed in parentheses, which is the *ID* number of the edit field where it should get the characters for the string variable *msg$*. In this case, we've specified edit field number 1 as the source of the data.

The last three statements verify that the contents of the edit field were indeed assigned to the string variable. First, a WINDOW statement activates document Window #1 so we can use it. (We've resized the document window so all the windows are visible on the screen.)

The "message" in our example is also relevant. The contents of an edit field are only available as a string. This is evident by the dollar sign at the end of the EDIT$ function name. If numeric information is to be entered via an edit field, we must use the VAL function to convert the string of numerals into a numeric variable.

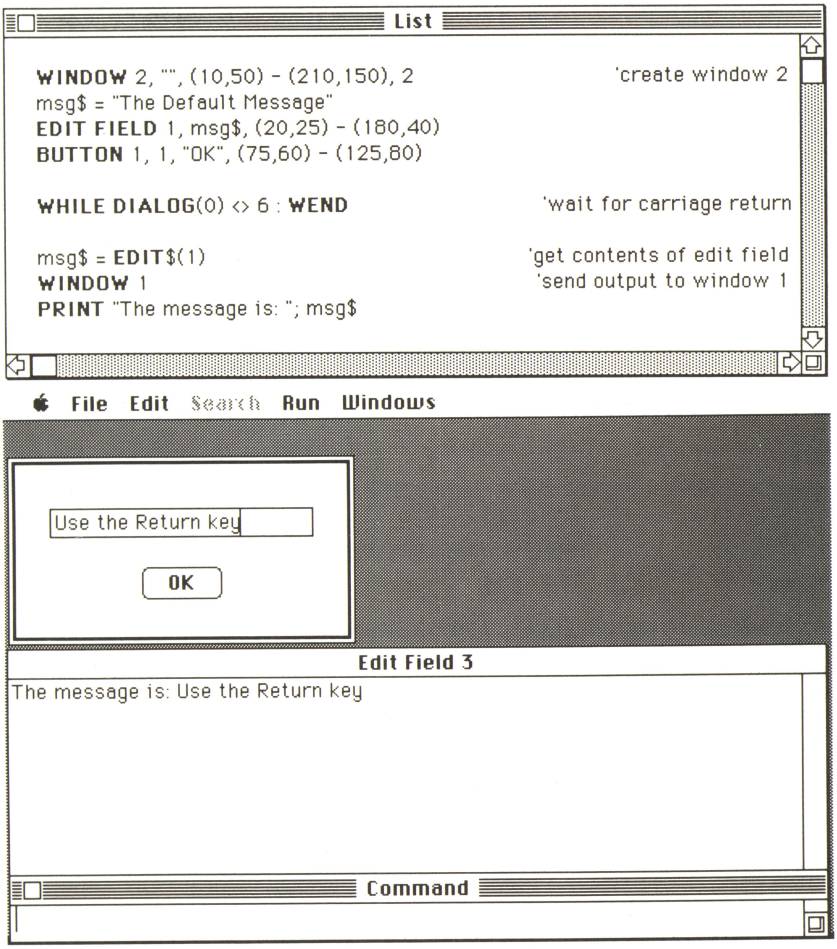
Using the Return key in an edit field

Our current program creates a button for the user to click when the contents of the edit field are to be accepted by the program. We did this because, at this point, pressing the Return key while typing in an edit field has no direct effect. But there is a way we can use the Return key to have the program accept the contents, as shown in Figure 14-19.

Pressing the Return key while typing in an edit field causes dialog event 6 to occur (see Figure 14-22 for a complete table of dialog events). Consequently, it's an easy matter to change the WHILE...WEND loop to check for the occurrence of dialog event 6, rather than event 1, which is caused by a button being pressed. So when we click the OK button now, nothing happens.

FIGURE 14-19

Using the Return key in an edit field



```
WINDOW 2, "", (10,50) - (210,150), 2              'create window 2
msg$ = "The Default Message"
EDIT FIELD 1, msg$, (20,25) - (180,40)
BUTTON 1, 1, "OK", (75,60) - (125,80)

WHILE DIALOG(0) <> 6 : WEND                 'wait for carriage return

msg$ = EDIT$(1)                            'get contents of edit field
WINDOW 1                                    'send output to window 1
PRINT "The message is: "; msg$
```

Use the Return key

```
The message is: Use the Return key
```

But most programs we work with on the Macintosh allow us to either press the Return key, or click a button. We could do this by simply modifying the WHILE statement like this:

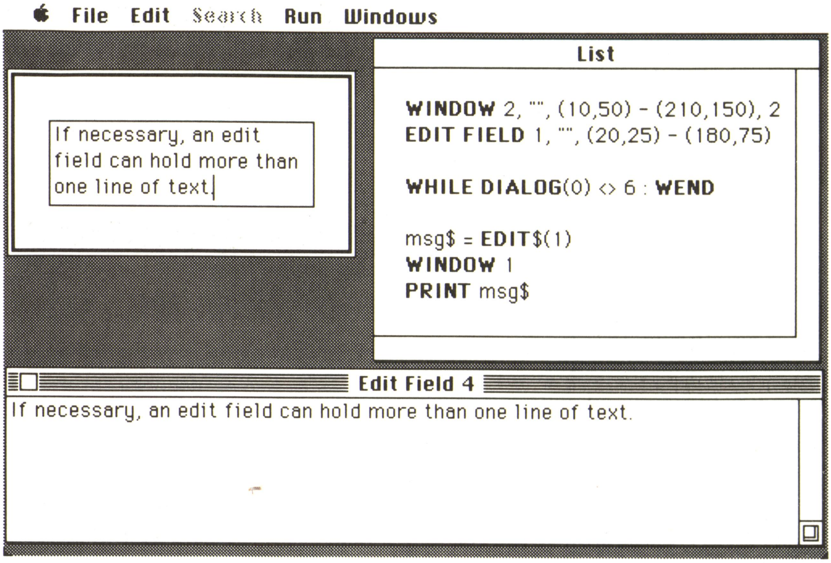
```
WHILE DIALOG(0) <> 1 AND DIALOG(0) <> 6
```

Using larger edit fields

Most of the time, the edit fields we require will be small. We may only want the user to enter something like the name of a program, or some other fairly short piece of information. Occasionally, we may want a larger edit field, capable of containing more data.

FIGURE 14-20

Using larger edit fields



```
WINDOW 2, "", (10,50) - (210,150), 2
EDIT FIELD 1, "", (20,25) - (180,75)

WHILE DIALOG(0) <> 6 : WEND

msg$ = EDIT$(1)
WINDOW 1
PRINT msg$
```

If necessary, an edit
field can hold more than
one line of text.

```
If necessary, an edit field can hold more than one line of text.
```

As we see in Figure 14-20, we can create an edit field with larger dimensions. When typing in this edit field, the user need not worry about reaching the end of a line. BASIC automatically "word wraps" to the next line. However, when the contents of the edit field are accepted by the program with the EDIT$ function, only one string of characters is stored, even though several lines of information may have been typed.

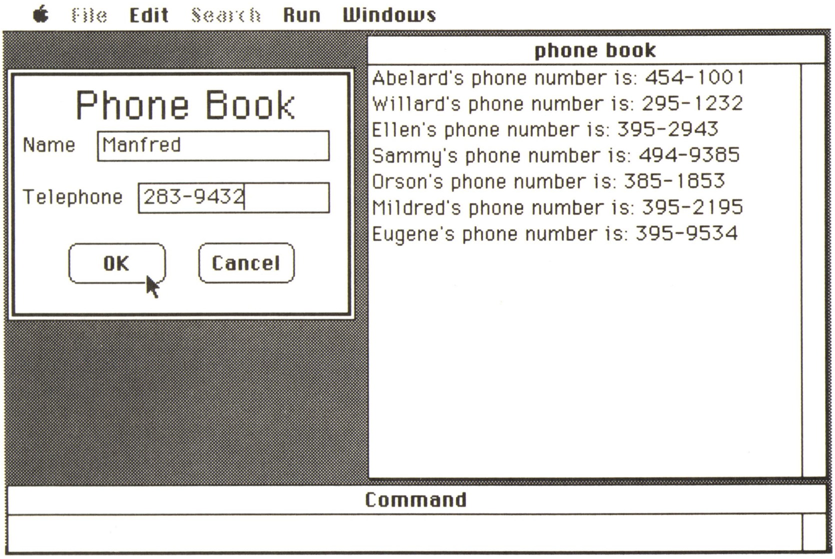
PUTTING-IT-ALL-TOGETHER PROGRAM

Now that we can create and use edit fields in our programs, let's proceed with a final example that incorporates windows, buttons, *and* edit fields. We'll also see how to switch back and forth between two edit fields.

As you can see in Figure 14-21, we've created a dialog box that could be used to maintain a computer phone book. This dialog box has two edit fields: one for the name and the other for the telephone number. It also has two buttons: one for OK and the other for Cancel. The program also displays the names and phone numbers which have been entered in the default Output window.

FIGURE 14-21

A dialog box with edit fields and buttons



Name Manfred

Telephone 283-9432

```
Abelard's phone number is: 454-1001
Willard's phone number is: 295-1232
Ellen's phone number is: 395-2943
Sammy's phone number is: 494-9385
Orson's phone number is: 385-1853
Mildred's phone number is: 395-2195
Eugene's phone number is: 395-9534
```

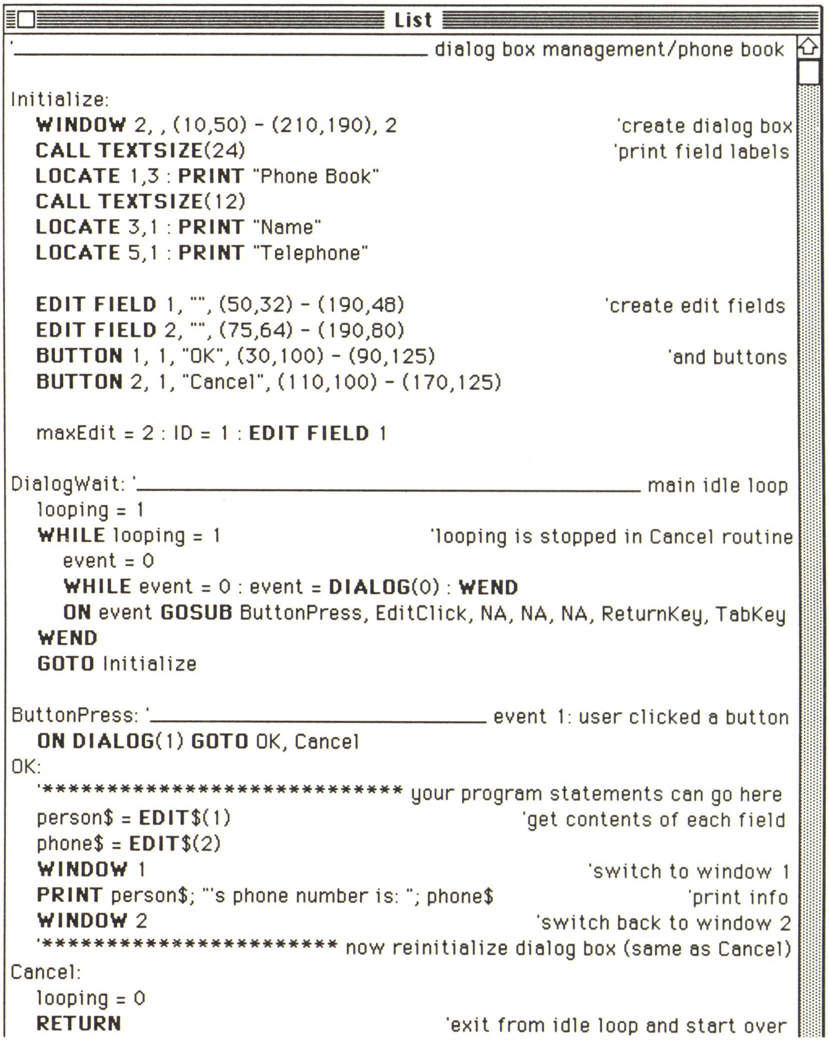
Before we look at the program that creates this dialog box, there is one thing we should consider. We are going to be using *two* edit fields in this program, and would like to allow the user to move back and forth between them in the usual manner. There are two ways the user can switch from one edit field to another in this program:

- Click the mouse pointer in an edit field.
- Press the Tab key.

Neither of these methods is automatic. They both cause dialog events that must be detected by our program if it is to handle this switching requirement. Therefore, we will be seeing a lot of the DIALOG function, which is how all this is handled. We'll explain all of this shortly, but now let's look at the program, shown in Figure 14-22 on the next page.

This example program is not a complete application. All it does is set up and maintain the dialog box that would be used for the input of a series of names and telephone numbers. Near the middle of the program you'll notice the remark, *Your program statements can go here.* This is where you might expand our example, in order to actually update a phone-book file on the disk.

The first section of the program, following the label *Initialize,* uses a WINDOW statement to create a bordered dialog box. Then, several text-output statements create the text that appears in the dialog box. Notice the two EDIT FIELD statements, and the two BUTTON statements that create the functional elements inside the dialog box.



```
'_________________________________ dialog box management/phone book

Initialize:
   WINDOW 2, , (10,50) - (210,190), 2                   'create dialog box
   CALL TEXTSIZE(24)                                    'print field labels
   LOCATE 1,3 : PRINT "Phone Book"
   CALL TEXTSIZE(12)
   LOCATE 3,1 : PRINT "Name"
   LOCATE 5,1 : PRINT "Telephone"

   EDIT FIELD 1, "", (50,32) - (190,48)                 'create edit fields
   EDIT FIELD 2, "", (75,64) - (190,80)
   BUTTON 1, 1, "OK", (30,100) - (90,125)                   'and buttons
   BUTTON 2, 1, "Cancel", (110,100) - (170,125)

   maxEdit = 2 : ID = 1 : EDIT FIELD 1

DialogWait: '_______________________________________ main idle loop
   looping = 1
   WHILE looping = 1                   'looping is stopped in Cancel routine
      event = 0
      WHILE event = 0 : event = DIALOG(0) : WEND
      ON event GOSUB ButtonPress, EditClick, NA, NA, NA, ReturnKey, TabKey
   WEND
   GOTO Initialize

ButtonPress: '___________________________ event 1: user clicked a button
   ON DIALOG(1) GOTO OK, Cancel
OK:
   '**************************** your program statements can go here
   person$ = EDIT$(1)                           'get contents of each field
   phone$ = EDIT$(2)
   WINDOW 1                                           'switch to window 1
   PRINT person$; "'s phone number is: "; phone$             'print info
   WINDOW 2                                      'switch back to window 2
   '********************** now reinitialize dialog box (same as Cancel)
Cancel:
   looping = 0
   RETURN                                'exit from idle loop and start over
```

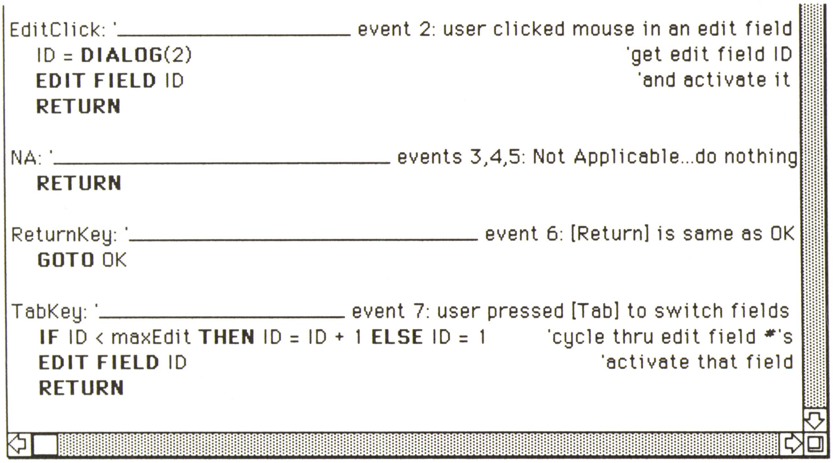
FIGURE 14-22

The phone-book program

(continued)

FIGURE 14-22

(continued)



```
EditClick: '____________________ event 2: user clicked mouse in an edit field
  ID = DIALOG(2)                                          'get edit field ID
  EDIT FIELD ID                                           'and activate it
  RETURN

NA: '____________________________ events 3,4,5: Not Applicable...do nothing
  RETURN

ReturnKey: '____________________________ event 6: [Return] is same as OK
  GOTO OK

TabKey: '___________________ event 7: user pressed [Tab] to switch fields
  IF ID < maxEdit THEN ID = ID + 1 ELSE ID = 1      'cycle thru edit field #'s
  EDIT FIELD ID                                      'activate that field
  RETURN
```

Since our program uses multiple edit fields, it needs to keep track of how many edit fields there are, done with the variable *maxEdit,* and which edit field is active, done with the variable *ID.* A short EDIT FIELD statement is used to activate edit field 1 just prior to the program's entering the idle loop section, where it loops until a dialog event occurs.

In this program a dialog event occurs when something happens in a dialog box and the program loops in the *DialogWait* section until something does. There are several different things that can happen. Figure 14-23 contains a complete table of all event possibilities and their corresponding values. Events 3 and 5 should be fresh in your memory, since we used them often when we worked with document windows, and we've also briefly mentioned events 6 and 1.

FIGURE 14-23

A table of dialog events and their values

Dialog Event	Description
1	User clicked a button
2	User clicked in an inactive edit field
3	User clicked in an inactive document window to make it active
4	User clicked the close box of a document window
5	User dragged a document window, uncovering part of another that will need refreshing
6	User pressed the Return key in an edit field to accept data
7	User pressed the Tab key to switch edit fields

When a dialog event occurs, its number is saved in the variable *event* and the program falls through to an ON...GOSUB statement. This statement directs the program to the appropriate subroutine, depending upon which dialog event occurred. There are four different dialog events that our program must respond to, since the three document-window events are not applicable here. Notice how efficiently the ON...GOSUB statement works in this capacity.

If dialog event 1 occurs, it means the user clicked a button, and the program proceeds to either the *OK* or *Cancel* sections. The *OK* section in this program gets the person's name and phone number, switches output to window 1 (a document window), and prints the information there. The program then switches back to window 2 (the dialog box), and moves on to the *Cancel* section. Here, the variable *looping* is set to 0 and the program is returned to the idle loop, where the WEND sends it to the WHILE. Since *looping* is no longer 1, the idle loop ends and the GOTO statement sends the program to the *Initialize* section, which reinitializes the dialog box.

If a dialog event 2 occurs, the user clicked in an inactive edit field in order to type in that field. Our program processes this dialog event in the *EditClick* subroutine. However, the program must activate the edit field before this is possible. The program can tell which edit field the user clicked by using the DIALOG(2) function, which returns the *ID* number of the edit field that was clicked. The program then activates the edit field with a short EDIT FIELD statement with only the *ID* number specified.

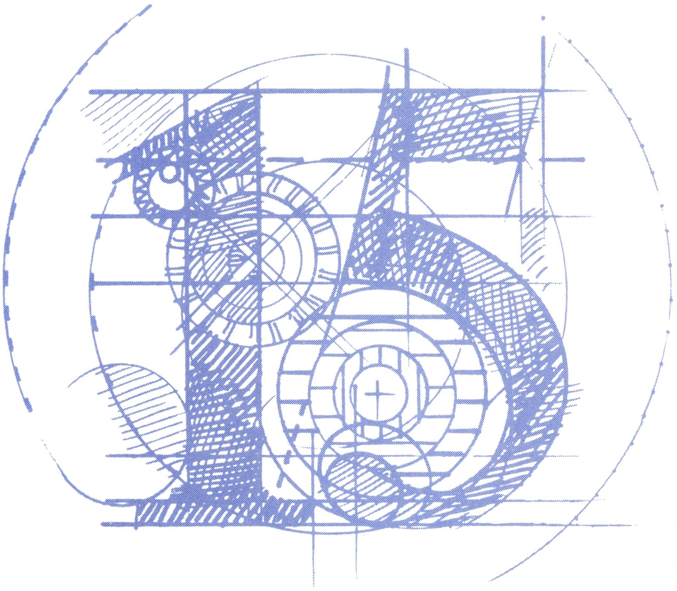
Even though we are relatively sure that the three *window* events will not occur in the program (we only handle button and edit field events), it's good programming practice to include someplace for the program to go to, just in case window events do happen to occur. That's the purpose of the label *NA:* where a RETURN statement simply sends control back to the *DialogWait* idle loop.

Besides clicking a particular edit field, pressing the Tab key is another common method of letting the user move from one edit field to the next. This is usually interpreted as moving to the "next" edit field in *ID* number sequence, or returning to edit field 1 when there are no further edit fields. Pressing the Tab key causes dialog event 7, which is processed by the BASIC statements in the *TabKey* subroutine. An IF...THEN...ELSE statement is used to either increment the edit field *ID* number, or reset it if all edit fields have been processed already. The short EDIT FIELD statement then activates the next edit field before the program returns to the *DialogWait* loop.

Later, if we wanted to add another edit field to our program, we would only have to create it with a third EDIT FIELD statement, and then change the value of the variable *maxEdit* accordingly.

REFRESH-YOURSELF WINDOW

In this chapter we've learned how to create windows of all types and sizes. We've seen that we can use more than one window in our programs, and direct output to any window we desire. Buttons and edit fields are an important part of windows used as dialog boxes, and we've seen how we can create and manage them in our BASIC programs. Most important, we've learned how to refresh the contents of a window, in case it has become uncovered by another window during the execution of our program.



QuickDraw

Earlier in the book we claimed that the Macintosh is an all-graphics *computer, and that there exists a powerful, underlying structure that gives the Mac its ability to represent everything with pixels. This "graphics structure" is actually a collection of programs permanently etched into the Mac electronics. Inside the Mac is a large* read-only memory *(commonly referred to as*

ROM) that contains a suite of over 400 powerful routines that perform specific services for us. A large portion of these services is devoted to drawing the Macintosh graphics, and these routines are referred to as *QuickDraw.* Although QuickDraw's routines are written in the language of the Macintosh's 68000 microprocessor, Microsoft has made many of them accessible directly from BASIC through the use of the CALL statement, which we will learn about shortly.

You might be wondering what QuickDraw offers that Microsoft BASIC's regular graphics statements, such as PSET, LINE, and CIRCLE don't. First, QuickDraw gives BASIC incredible text-handling abilities. We can print text in any font that is on the disk, in any combination of eight different textfaces, in four different modes, and in many different point sizes. Instead of lines just one pixel wide, which normal BASIC offers with PSET and LINE, we can draw lines of almost any width and height with the QuickDraw *pen.* This pen can draw in black, white, or gray, or with an unbelievable number of custom patterns. We can draw ordinary circles and rectangles, like those created with regular BASIC statements. But with QuickDraw we can also draw rectangles with rounded corners, and polygons with multiple sides. Finally, we can *fill* any QuickDraw shape with a pattern, which is not possible with regular BASIC statements like CIRCLE. If you are familar with MacPaint, you might like to know that it is completely built around QuickDraw. In fact, we could write a BASIC program that would work like MacPaint.

WHAT IS QUICKDRAW?

The QuickDraw routines, designed by Bill Atkinson of Apple Computer, are the very heart of the Macintosh architecture. Every major program that follows the standard Apple graphics-style uses them. In fact, every single activity that takes place on the Mac screen must flow through QuickDraw! (Don't worry, the QuickDraw part of the Macintosh ROM will not overheat!)

The name QuickDraw is Atkinson's way of saying the routines are really "quick"—in fact these routines are faster than any graphics seen before on personal computers. Because of this high speed, it is possible to do things graphically on the Macintosh that on previous microcomputers were extremely difficult, too slow, too crude, or just plain impossible.

You may be wondering if the PSET, LINE, and CIRCLE graphics statements we learned already are part of QuickDraw. The answer is no. Instead, PSET, LINE, and CIRCLE work the same as statements found in the lineage of Microsoft BASIC language products that have been around since personal computing started. The PSET, LINE, and CIRCLE statements, for example, are found in Microsoft's IBM PC BASIC. By including long-established graphics statements in the Macintosh version of BASIC, program conversion from IBM PC BASIC to Macintosh Microsoft BASIC is easier. But even though these statements are not part of the Microsoft BASIC QuickDraw vocabulary that we are about to learn on the Macintosh, they *do* make use of the QuickDraw routines stored in the ROM. This interaction remains invisible to us.

If you plan to convert a Macintosh BASIC program to run on an IBM PC, you won't want to use QuickDraw statements, since they are specific to the Macintosh. But if you don't plan to convert your Macintosh program to run on an IBM PC, go ahead and use QuickDraw with relish.

ACCESSING QUICKDRAW FROM BASIC WITH CALL

The question you're probably asking is, "How do I get these great QuickDraw routines into my Microsoft BASIC programs?" The answer is simple: with the CALL statement. Generally, the CALL statement is used to access any programs that are written in the language of the Macintosh 68000 microprocessor, or *machine language.* CALL transfers control from BASIC to the machine-language program. When the machine-language program is finished, it returns control back to the BASIC program.

Not all machine-language programs are in the ROM. Programs are written in machine language by advanced programmers when a BASIC program or subroutine cannot easily accomplish a certain function. Faster execution is often a reason machine language is used, or when control of some external or internal hardware device is required.

Since QuickDraw is a collection of machine-language routines in the ROM, the CALL statement provides a convenient way of accessing them. To access QuickDraw machine-language routines, you simply put the CALL statement in front of the name of the QuickDraw function you want to use. You did this earlier in Chapter 4, when you used the TEXTSIZE statement: For example, *CALL TEXTSIZE(14)* causes the QuickDraw machine-language function that changes textsize to be executed. The value 14 is called a *QuickDraw parameter.*

CALL is an optional statement. But if CALL is omitted, the parentheses must also be omitted. We could write the previous statement without CALL as *TEXTSIZE 14* and it would work exactly like the first.

But even though CALL is optional, we should always use it because without CALL it's too easy to write a QuickDraw statement that looks like an alphanumeric label, which can be confusing.

Now that we know how QuickDraw routines are called into our BASIC programs, we're ready to look at the individual functions. Since we've already worked a bit with TEXTSIZE in previous chapters, let's begin by taking a closer look at it and the other QuickDraw text functions.

THE QUICKDRAW TEXTSIZE STATEMENT

We learned in Chapter 4, "Introduction to Graphics," that we can change the size of fonts with the TEXTSIZE call, like this:

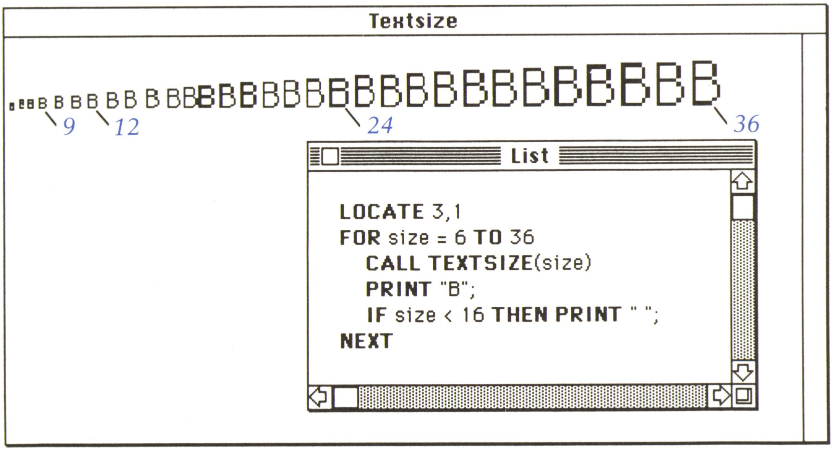
CALL TEXTSIZE(*n*)

where *n* in parentheses specifies the *point size* of the font and can be any number between 2 and 127 (inclusive). The term *point* comes from the printing world: A point represents 1⁄72 inch. Thus a character in 72-point type is about 1 inch high. Likewise, 36-point type is ½ inch high, 18 point is ¼ inch, and so on.

Type in and run the program shown in Figure 15-1. It causes the letter *B* to be printed on the screen in textsizes ranging from 6- to 36-point using the default Geneva font.

Font scaling

Notice in Figure 15-1 that some *B*s look a lot better than others. What is happening here is that the Microsoft BASIC disk does not contain a file of characters for the Geneva font in every possible font size. In fact, this disk contains two sets of Geneva: one in 12 point and one in 9 point. When Microsoft BASIC can't find the point size we are asking for, it designs the character for us by scaling the image of a character that is available up or down. A character that comes from a size actually stored on the disk looks better than a character that must be scaled. A scaled character that is an even multiple of an available point size still looks pretty good, as you can see with point size 24 in the example, and a character that must be scaled without an even multiple looks only fair. Some font sizes scale rather poorly, and as the point size gets larger this distortion can get worse, as you can see in the examples between 28- and 36-point.



```
LOCATE 3,1
FOR size = 6 TO 36
   CALL TEXTSIZE(size)
   PRINT "B";
   IF size < 16 THEN PRINT " ";
NEXT
```

FIGURE 15-1

Changing the textsize

When talking about textsizes in both Chapter 4 and in this section, we alluded several times to the fact that the Macintosh can use different *fonts.* Let's see what this means to us as BASIC programmers.

THE QUICKDRAW TEXTFONT STATEMENT

Fonts, as you probably already know, are character sets that have specific typefaces, or style of type. People familar with MacWrite, MacPaint, or Microsoft Word know some of the standard Macintosh fonts, which go by names like Chicago, Geneva, New York, and so on. Each font has some *particular idea* behind it: Chicago is a bold, strong font; New York is a proportionally spaced, newspaper-like font with serifs; Monaco is a monospaced font; and so on.

Fonts are selected in Microsoft BASIC with the TEXTFONT statement as follows:

CALL TEXTFONT(*n*)

where *n* is usually any number from 0 to 12. Each number stands for one of 12 common Apple fonts that the Macintosh can display. (There are in fact more than 12: Apple has designed some new fonts and others are supplied by other vendors. But we won't go into these other fonts here.) The table in Figure 15-2 on the next page shows which font is selected by each number in the TEXTFONT statement. Additional information regarding font sizes is also given in this table. Each font comes in different sizes: 9, 10, 12, 14, 18, or 24 points. Each of these font files has been carefully designed by Apple to contain the best arrangement of pixels for its size.

FIGURE 15-2

The fonts that can be accessed with TEXTFONT and their available sizes

		Available Point Sizes						
(n)	Font	9	10	12	14	18	20	24
0	Chicago			◇ ◆				
1	Geneva	◇ ◆	◇	◇ ◆	◇	◇	◇	◇
2	New York	◇	◇	◇	◇	◇	◇	◇
3	Geneva (same as font 1)							
4	Monaco	◇ ◆		◇				
5	Venice				◇			
6	London					◇		
7	Athens					◇		
8	San Francisco					◇		
9	Toronto	◇		◇	◇	◇		◇
10	Seattle							
11	Cairo					◇		
12	Los Angeles			◇				◇

◆ Fonts available on Microsoft BASIC Disk
◇ Fonts available in standard Macintosh System Disk Fonts file

Once you have selected a particular font with TEXTFONT, it stays in operation until the next TEXTFONT statement changes it. Therefore the statements:

```
CALL TEXTFONT(0)
PRINT "Climb Every Mountain"
:
[intervening program lines]
:
CALL TEXTFONT(1)
PRINT "Follow Every Shore"
:
[intervening program lines]
:
END
```

print *Climb Every Mountain* and everything after it in Chicago font, and *Follow Every Shore* and everything after it in Geneva font.

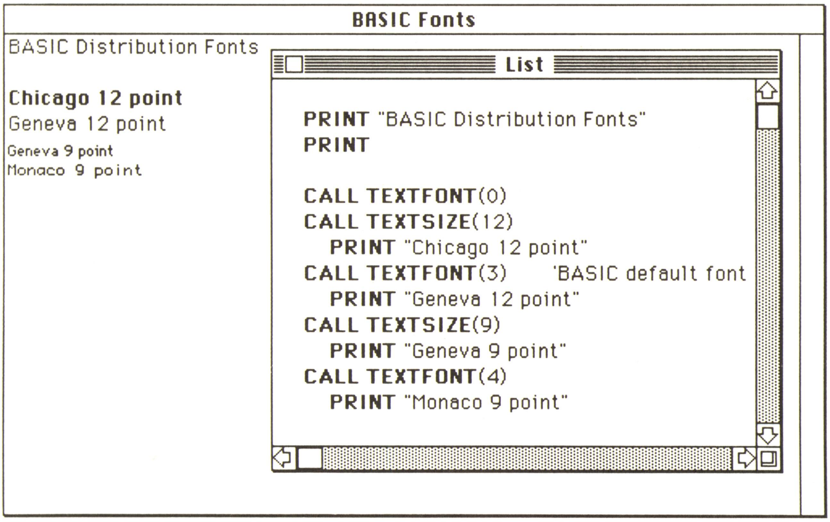
Fonts provided on the BASIC disk

Before we get too much further, you need to understand where fonts come from. When you purchase Microsoft BASIC for the Macintosh, the disk comes with its own system folder, which allows the BASIC disk to be self-starting. Contained in this folder are special files that contain the pixel definitions of the fonts and sizes provided on the disk. These fonts and sizes have been custom designed by artists to optimize these pixel patterns so that the letters look smooth and tailored.

However, this system folder does not contain *all* the fonts that BASIC is capable of displaying. The Microsoft BASIC disk comes with only three fonts: Chicago, Geneva, and Monaco. Geneva, as we've mentioned, comes in two sizes: 9 point and 12 point; Monaco is provided in 9 point; and Chicago only in 12 point. This is indicated in the table in Figure 15-2, and the fonts themselves can be seen in Figure 15-3.

Why so few fonts on the Microsoft BASIC disk? The reason is space limitations. Each font file consumes a considerable number of bytes on the disk. Including all the fonts would consume so much space that there would be little room left for the Sample Programs, not to mention the decimal version of the BASIC Interpreter.

There are two ways to make additional fonts available to Microsoft BASIC. We can replace the disk's system folder with a folder containing the fonts we wish to use. But since we've created separate disks for each version of BASIC, we do have room to actually copy other fonts and sizes onto our working disks. Doing this is simply a matter of using the *Font Mover* program and *Fonts* file provided on the system disk you received with your Macintosh. The Fonts file contains the extra Apple fonts, such as Cairo, that are not included in the system folders on either the system or BASIC disks. The first step is to copy both the Font Mover program and



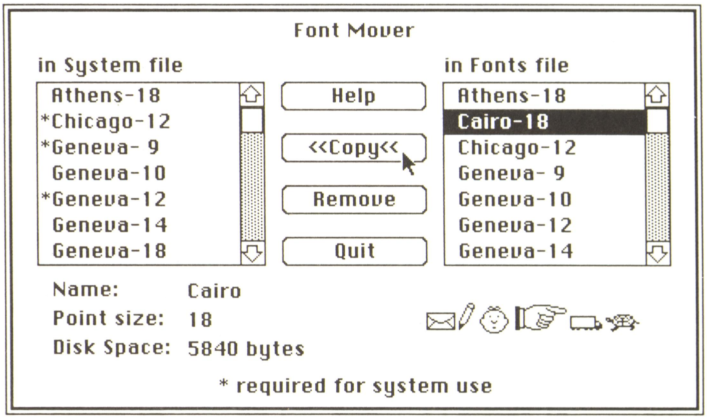
```
BASIC Distribution Fonts

Chicago 12 point
Geneva 12 point
Geneva 9 point
Monaco 9 point
```

```
PRINT "BASIC Distribution Fonts"
PRINT

CALL TEXTFONT(0)
CALL TEXTSIZE(12)
  PRINT "Chicago 12 point"
CALL TEXTFONT(3)      'BASIC default font
  PRINT "Geneva 12 point"
CALL TEXTSIZE(9)
  PRINT "Geneva 9 point"
CALL TEXTFONT(4)
  PRINT "Monaco 9 point"
```

FIGURE 15-3

The fonts available on the Microsoft BASIC disk

Fonts file from the system disk onto our BASIC disk by dragging their icons from the system desktop to the BASIC desktop. Next, we double-click the Font Mover icon to open this window:


Font Mover
in System file
Athens-18
*Chicago-12
*Geneva- 9
Geneva-10
*Geneva-12
Geneva-14
Geneva-18
Help
<<Copy<<
Remove
Quit
in Fonts file
Athens-18
Cairo-18
Chicago-12
Geneva- 9
Geneva-10
Geneva-12
Geneva-14

Name: Cairo
Point size: 18
Disk Space: 5840 bytes

* required for system use

To transfer a font from the Fonts-file window to the BASIC disk's System-file window so we can use it in our programs, we select the font we want (such as Cairo, in the Font Mover window shown above), then click the Copy button. The disk spins while the Mac copies the font into the BASIC system file, and then we can click the Quit button to return to the BASIC desktop. Next, we trash both Font Mover and the Fonts file to free up their space on our disk (but before you do this, *make sure* you're on the BASIC desktop and not the system disk's desktop). Consult the Macintosh manual for complete instructions on moving fonts.

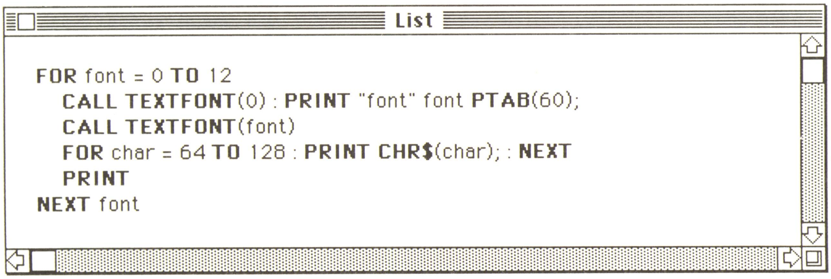
Playing with fonts

Let's suppose that we have copied all of the fonts available in the Fonts file into our BASIC disk's system folder. If we enter and run the program shown in Figure 15-4, it will display all the fonts available to Microsoft BASIC. As it operates you will hear the disk spinning a lot, because each font is taken from the system folder as the program requests it. You should get the results shown in the Output window.

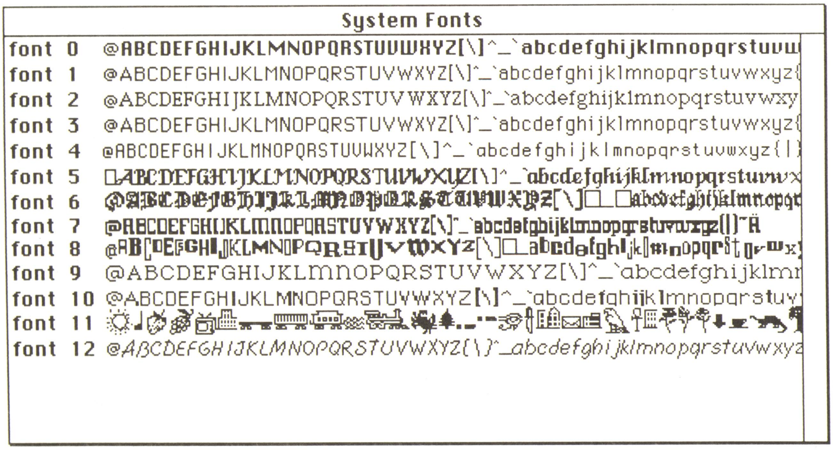
If you run the program and find that some of the fonts are replaced by the Geneva font, this means that your system folder did *not* contain all the fonts the program requested. What BASIC does when it can't find a font file specified by a TEXTFONT statement is substitute the Geneva font. If you use the original BASIC disk with the program in Figure 15-4, the Geneva font will be substituted for all fonts except Chicago and Monaco.

FIGURE 15-4

The fonts available with the Macintosh system disk



```
FOR font = 0 TO 12
   CALL TEXTFONT(0) : PRINT "font" font PTAB(60);
   CALL TEXTFONT(font)
   FOR char = 64 TO 128 : PRINT CHR$(char); : NEXT
   PRINT
NEXT font
```



```
font 0   @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvw
font 1   @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{
font 2   @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy
font 3   @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{
font 4   @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
font 5   □ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwx
font 6   @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]□_□abcdefghijklmnopq
font 7   @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~Ä
font 8   @AB[DEFGHIJKLMNOPQRSTUVWXYZ[\]□_`abcdefghijklmnopqrstuvwx
font 9   @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
font 10  @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuv
font 11
font 12  @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
```

Note that fonts 1 and 3 are the same. Font 3 is the *application font,* the font used by BASIC for program listings and default output to the Output window. Geneva font was chosen by Microsoft as the default font over other fonts because it offers the best of the most important compromises. It is not as dark as Chicago, which you may have noticed Apple used for the Mac's menus, dialog boxes, and other labeled elements, and it's not too fancy like New York or Toronto with their serifs.

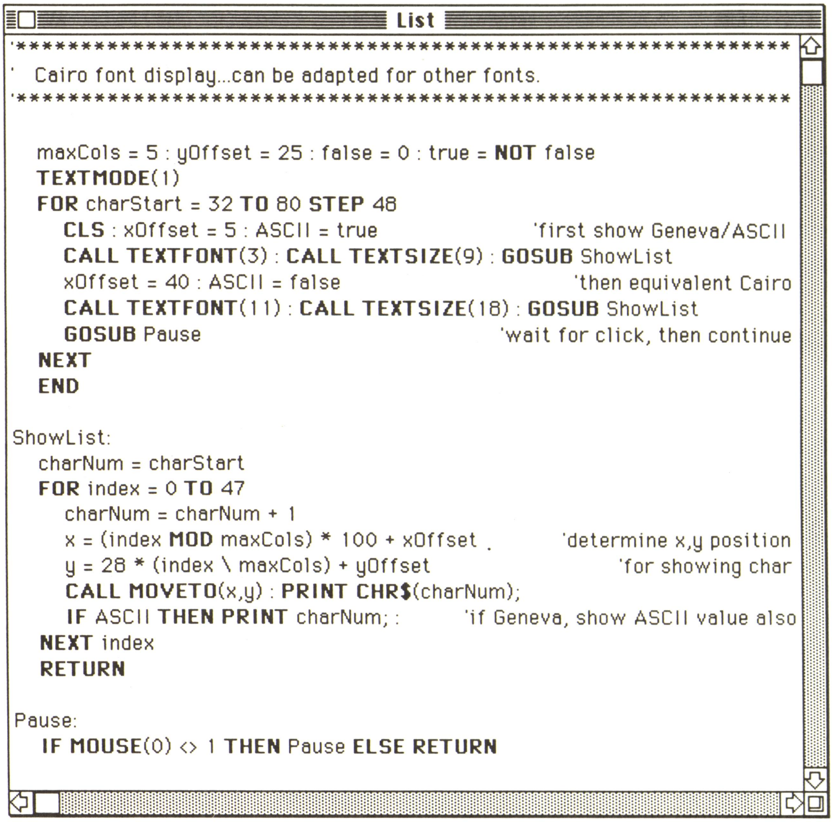
The little rectangle preceding the first character in font 5 (Venice) is printed if a character is not defined in the font file. Here, for example, the @ symbol does not exist in Venice, the ^ and ` symbols don't exist in font 6 (London), and ^ doesn't exist in font 8 (San Francisco).

The Cairo font

If you like Macintosh graphics, you're probably dying to know more about font 11, the Cairo font. Cairo produces hieroglyphic characters reminiscent of the symbols found on the tombs and pyramids of Egypt. If you like games and icons, you'll love Cairo. Because QuickDraw treats text as fast as it does pure graphics, you can build fast-paced action games around the Cairo font. You may also find ways to use Cairo in more industrial-strength programs—as small icons, perhaps.

In order to see the symbols that are available with Cairo, and also to see what each Cairo character corresponds to in the familiar Geneva font as well as their ASCII equivalents, type in the program shown below in Figure 15-5. The program displays the Cairo symbols in two passes. The first 48 Cairo characters are displayed on the screen next to their Geneva or ASCII equivalents, then pressing the mouse button causes the next 48 characters to be displayed. We can use the output of the program, shown in Figure 15-6, to determine which Geneva character or ASCII number to put in PRINT statements to get the desired Cairo symbol. And we can hold down the Command-Shift-4 keys all at once to dump the screen contents to our printer and get a hard copy.

There is a special point to keep in mind when using two different fonts like we are in this program. When we first wrote the program, we



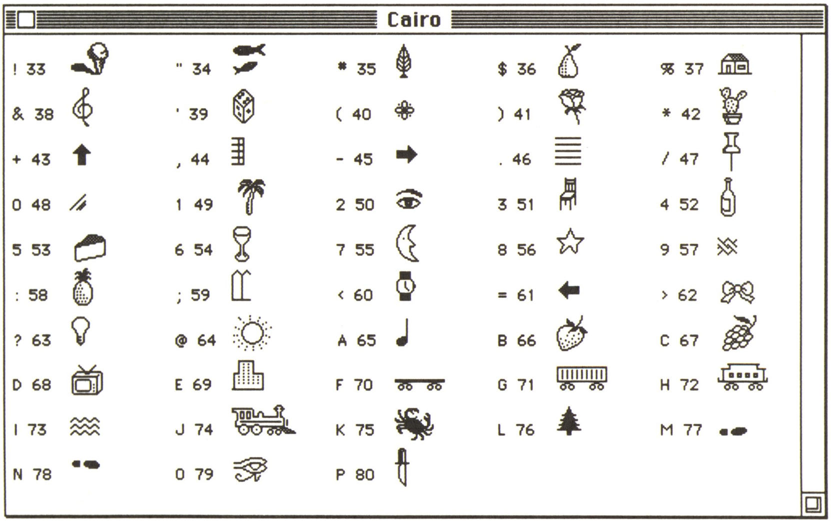
```
'**********************************************************
'  Cairo font display...can be adapted for other fonts.
'**********************************************************

   maxCols = 5 : yOffset = 25 : false = 0 : true = NOT false
   TEXTMODE(1)
   FOR charStart = 32 TO 80 STEP 48
      CLS : xOffset = 5 : ASCII = true                  'first show Geneva/ASCII
      CALL TEXTFONT(3) : CALL TEXTSIZE(9) : GOSUB ShowList
      xOffset = 40 : ASCII = false                        'then equivalent Cairo
      CALL TEXTFONT(11) : CALL TEXTSIZE(18) : GOSUB ShowList
      GOSUB Pause                                       'wait for click, then continue
   NEXT
   END

ShowList:
   charNum = charStart
   FOR index = 0 TO 47
      charNum = charNum + 1
      x = (index MOD maxCols) * 100 + xOffset            'determine x,y position
      y = 28 * (index \ maxCols) + yOffset                 'for showing char
      CALL MOVETO(x,y) : PRINT CHR$(charNum);
      IF ASCII THEN PRINT charNum; :             'if Geneva, show ASCII value also
   NEXT index
   RETURN

Pause:
   IF MOUSE(0) <> 1 THEN Pause ELSE RETURN
```

FIGURE 15-5

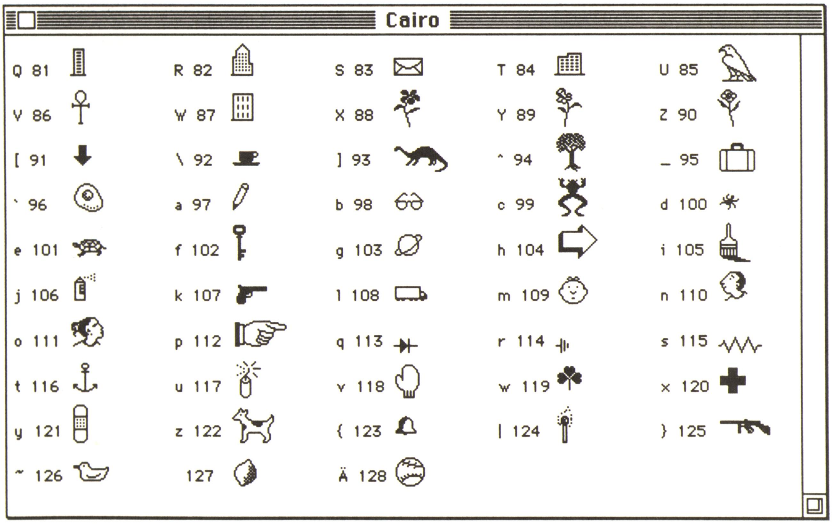
A program to generate a table of the Geneva and Cairo character sets and their ASCII codes

FIGURE 15-6

The table of Geneva and Cairo characters output by the font-display program



```
! 33      " 34      # 35      $ 36      % 37
& 38      ' 39      ( 40      ) 41      * 42
+ 43      , 44      - 45      . 46      / 47
0 48      1 49      2 50      3 51      4 52
5 53      6 54      7 55      8 56      9 57
: 58      ; 59      < 60      = 61      > 62
? 63      @ 64      A 65      B 66      C 67
D 68      E 69      F 70      G 71      H 72
I 73      J 74      K 75      L 76      M 77
N 78      O 79      P 80
```

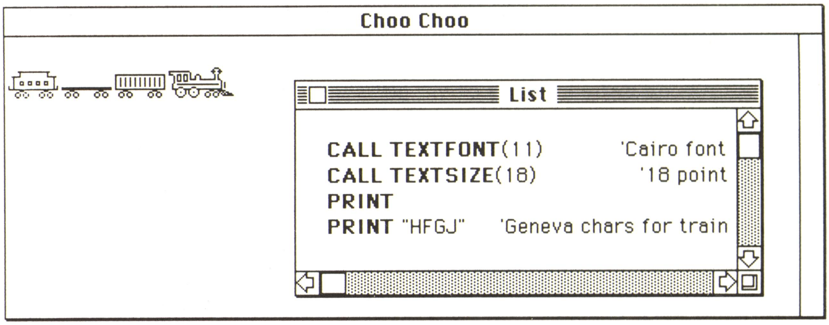


```
Q 81      R 82      S 83      T 84      U 85
V 86      W 87      X 88      Y 89      Z 90
[ 91      \ 92      ] 93      ^ 94      _ 95
` 96      a 97      b 98      c 99      d 100
e 101     f 102     g 103     h 104     i 105
j 106     k 107     l 108     m 109     n 110
o 111     p 112     q 113     r 114     s 115
t 116     u 117     v 118     w 119     x 120
y 121     z 122     { 123     | 124     } 125
~ 126       127     Ä 128
```

switched between displaying a Geneva character, then displaying its Cairo equivalent, and so on. But each time BASIC calls a new textfont with CALL TEXTFONT it causes the entire font to be read off the disk and into the Macintosh's memory. Rapid changes between two or more fonts result in considerable slowing of a program, since each font must be read in from the disk and the disk is slow. The way to get around this is to pull a font into memory and then use it as much as you can before changing to a new font. Thus, we rewrote the program so it displays all the Geneva characters, then without erasing the screen, goes back and displays all the Cairo characters. This speeded the program up by a factor of about 20 to 1.

FIGURE 15-7

A program that generates a series of Cairo symbols



```
CALL TEXTFONT(11)          'Cairo font
CALL TEXTSIZE(18)           '18 point
PRINT
PRINT "HFGJ"      'Geneva chars for train
```

Note that although the Cairo font is optimized at 18 points, we can use the TEXTSIZE statement to expand it to make larger symbols better suited for games.

The tables shown in Figure 15-6 will come in handy when we want to print a Cairo symbol and need to know which Geneva symbol to put in the PRINT statements. An example is shown in Figure 15-7, which creates a train from four of the Cairo symbols.

THE SPECIAL SYMBOLS

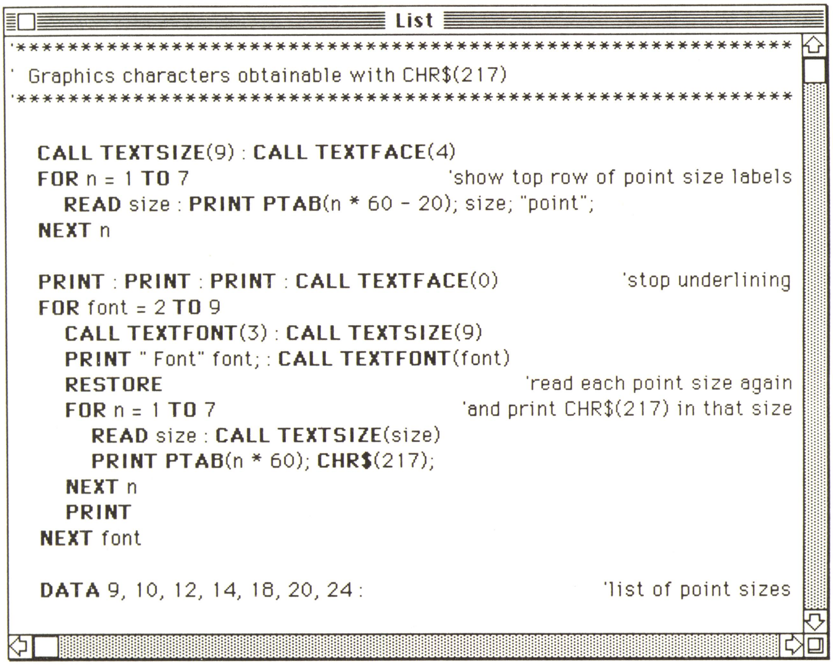
Apple, in all its wisdom, set aside one special ASCII code in each font to hold a graphics symbol. This symbol is accessed from the keyboard by pressing the Option-Shift-Tilde (~) keys together. From BASIC we can access it with CHR$(217). Figure 15-8 shows a program that displays all the special graphics symbols available in the Apple fonts.

Note that the fonts in color in Figure 15-8 are the ones that actually exist in a font file, and the others are simply scaled from them. We reproduced them all because even the scaled characters are useful. Fonts 10, 11, and 12 are not included in the table because they do not have special symbols.

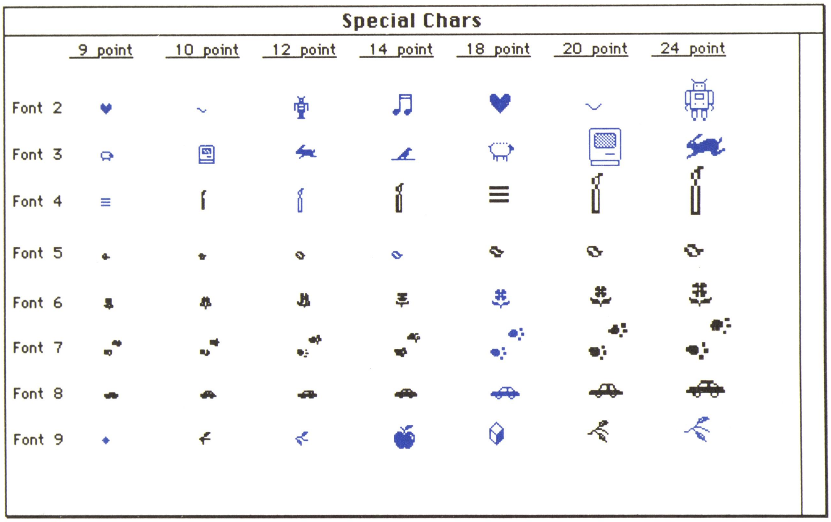
Font 0 (Chicago) has three special symbols that are not accessed with CHR$(217). These symbols are shown in Figure 15-9 on page 352. The Command-key character can be printed with CHR$(17), the check mark with CHR$(18), and the Apple with a bite out of it from the Apple menu with CHR$(20).

FIGURE 15-8

The special graphics characters program



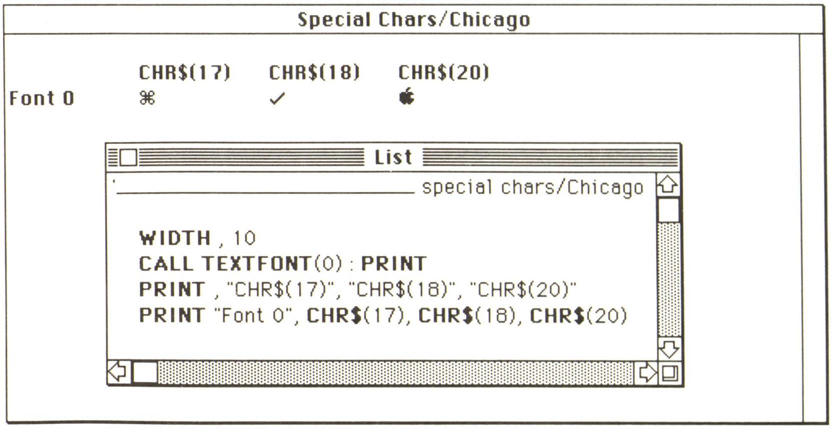
```
'*****************************************************************
' Graphics characters obtainable with CHR$(217)
'*****************************************************************

  CALL TEXTSIZE(9) : CALL TEXTFACE(4)
  FOR n = 1 TO 7                          'show top row of point size labels
    READ size : PRINT PTAB(n * 60 - 20); size; "point";
  NEXT n

  PRINT : PRINT : PRINT : CALL TEXTFACE(0)              'stop underlining
  FOR font = 2 TO 9
    CALL TEXTFONT(3) : CALL TEXTSIZE(9)
    PRINT " Font" font; : CALL TEXTFONT(font)
    RESTORE                                   'read each point size again
    FOR n = 1 TO 7                        'and print CHR$(217) in that size
      READ size : CALL TEXTSIZE(size)
      PRINT PTAB(n * 60); CHR$(217);
    NEXT n
    PRINT
  NEXT font

  DATA 9, 10, 12, 14, 18, 20, 24 :                     'list of point sizes
```



```
           9 point   10 point   12 point   14 point   18 point   20 point   24 point

Font 2

Font 3

Font 4

Font 5

Font 6

Font 7

Font 8

Font 9
```

FIGURE 15-9

The special graphics characters available for font 0 (Chicago)



```
        CHR$(17)    CHR$(18)    CHR$(20)
Font 0  ⌘           ✓           
```

```
'________________________ special chars/Chicago

WIDTH , 10
CALL TEXTFONT(0) : PRINT
PRINT , "CHR$(17)", "CHR$(18)", "CHR$(20)"
PRINT "Font 0", CHR$(17), CHR$(18), CHR$(20)
```

Now that we have explored the nuances of the TEXTSIZE and TEXTFONT QuickDraw routines, we're well prepared to tackle the TEXTMODE QuickDraw routine which lets us mix text and graphics in different combinations to suit our needs.

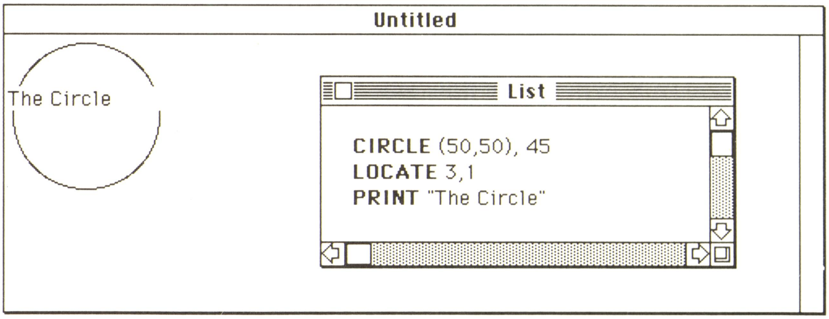
THE QUICKDRAW TEXTMODE STATEMENT

Programmers often mix text and numeric data with screen graphics. Mixing text and graphics occurs when we want to label a graph, put dollar amounts into a fixed grid, number rulers, display game characters on complex backgrounds, and so on. As an example, suppose we want to draw a circle and print the words *The Circle* in the middle of it. Sounds simple, doesn't it? Figure 15-10 shows a program that will do this using the tools we have learned up to this point: CIRCLE, LOCATE, and PRINT.

Look at what happened: The text did funny things to the pixels that make up the circle. Where the text starts it seems to have blotted out a portion of the circle. Also a section of the circle to the right of the text is

FIGURE 15-10

A program demonstrating how the PRINT statement destroys anything underneath where it prints



```
CIRCLE (50,50), 45
LOCATE 3,1
PRINT "The Circle"
```

missing. The reason this happened here is that BASIC prints text on the background in such a way that any existing black pixels underneath the text are changed to white pixels. In addition, BASIC acts like it is adding white spaces to the end of any text it prints, which is why it wipes out part of the circle on the right. Before you get too excited, there is a way out of this: the QuickDraw *textmodes.*

Microsoft BASIC can print text to the Output window in one of these four modes:

Value	Mode
0	Copy
1	OR
2	XOR
3	BIC

The statement used to control these modes is:

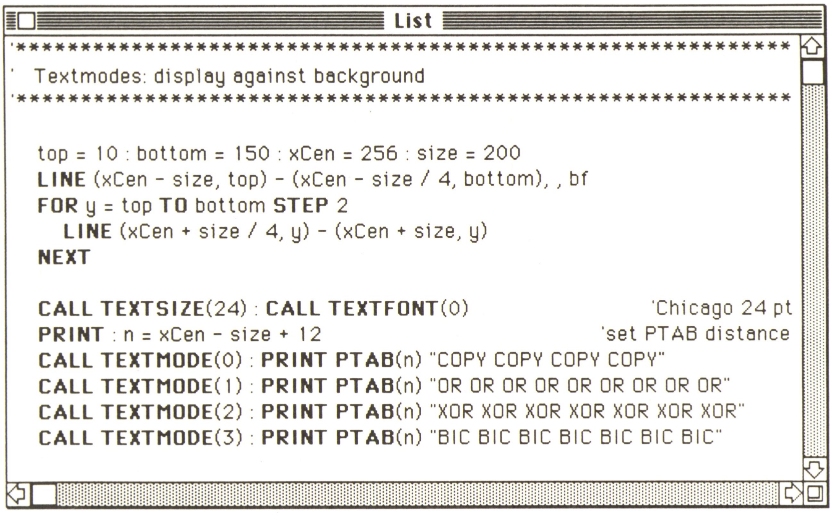
CALL TEXTMODE(*n*)

where *n* corresponds to one of the four text modes 0 through 3. An example of the use of TEXTMODE is shown in Figure 15-11, which contains a program that displays each of the four textmodes on a black, white, and patterned background.

Let's look at each of these modes individually and get a clearer idea of how they work with background patterns and graphics.

FIGURE 15-11

The four textmodes on different backgrounds

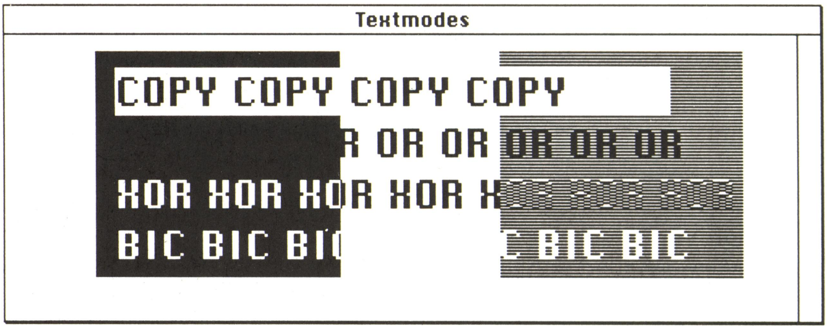


```
'**********************************************************
'  Textmodes: display against background
'**********************************************************


  top = 10 : bottom = 150 : xCen = 256 : size = 200
  LINE (xCen - size, top) - (xCen - size / 4, bottom), , bf
  FOR y = top TO bottom STEP 2
    LINE (xCen + size / 4, y) - (xCen + size, y)
  NEXT

  CALL TEXTSIZE(24) : CALL TEXTFONT(0)                    'Chicago 24 pt
  PRINT : n = xCen - size + 12                         'set PTAB distance
  CALL TEXTMODE(0) : PRINT PTAB(n) "COPY COPY COPY COPY"
  CALL TEXTMODE(1) : PRINT PTAB(n) "OR OR OR OR OR OR OR OR OR"
  CALL TEXTMODE(2) : PRINT PTAB(n) "XOR XOR XOR XOR XOR XOR XOR"
  CALL TEXTMODE(3) : PRINT PTAB(n) "BIC BIC BIC BIC BIC BIC BIC"
```

(continued)

FIGURE 15-11

(continued)



TEXTMODE(0): The Copy mode

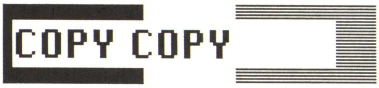
When BASIC is first started it defaults to TEXTMODE(0), called the *Copy* mode. In the Copy mode each character in a PRINT statement can be thought of as being surrounded by a little white box. When a letter is printed onto a background of graphics, every pixel in the box (both black and white) is copied directly onto the background, effectively overlaying all the pixels and destroying the background under the box. This is why the circle was erased a little above and below the beginning of the letter *T.* The "white box" extends about five spaces beyond the last character of a printed line, so part of the right side of the circle was also erased.

Each textmode overlays or "mixes" its printed characters with the background pattern in a unique way, which can be illustrated with a table of rules. Figure 15-12 shows the rule table and an example of the result for the Copy mode. The symbol ○ represents a white pixel and ● represents a black pixel.

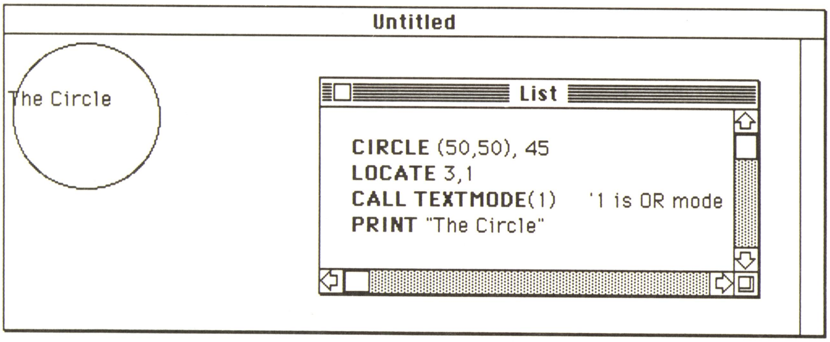
As the table indicates, the pixels in text characters take precedence over those in the background and, in effect, cover them up. So the way around the situation with our circle example shown in Figure 15-10 is to try a different textmode.

FIGURE 15-12

The pixel rules for TEXTMODE(0): The Copy mode



Text Pixel	Background Pixel	Final Pixel
○	○	○
○	●	○
●	○	●
●	●	●



```
CIRCLE (50,50), 45
LOCATE 3,1
CALL TEXTMODE(1)     '1 is OR mode
PRINT "The Circle"
```

FIGURE 15-13

The labeled-circle program with TEXTMODE(1): The OR mode

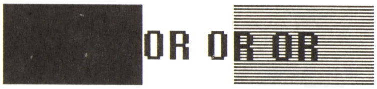
TEXTMODE(1): The OR mode

To change the textmode in our current program, we can simply change the number in parentheses in the TEXTMODE statement. Let's change the *0* to a *1* and run the program shown in Figure 15-13.

Look closely at the Output window of Figure 15-13 and you'll see that the text no longer changes all the black pixels to white, and that the text is overlaid on the circle background. What TEXTMODE(1) did is change the mode from Copy to what is called the *OR* mode. We have seen the word OR before, when we studied the IF...THEN statement. Then it was used for comparing different values or strings. In a similar way OR can be used for controlling how pixels act when they are combined. Figure 15-14 shows how the OR rules work.

We can see from the table that when the text is a white pixel, but the background is a black pixel, then the final pixel is black. If, on the other hand, the text is a black pixel and the background is a white pixel, the final pixel is still black. This is often referred to as the "black wins" mode. The name OR comes from the fact that *if pixel A OR pixel B* is black, then the background is black.

The OR mode is very useful when you want the background graphics to "show through" the text you are printing. If you study the printed text in Figure 15-13 you'll see that the circumference of the circle



Text Pixel	Background Pixel	Final Pixel
○	○	○
○	●	●
●	○	●
●	●	●

FIGURE 15-14

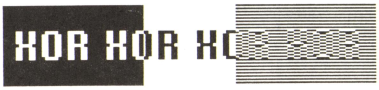
The pixel rules for TEXTMODE(1): The OR mode moves smoothly into the letter *T.* We will see the use of OR in other situations later in this chapter.

TEXTMODE(2): The XOR mode

The *XOR,* or *exclusive OR,* mode is a particularly useful mode. In the XOR mode, the pixels of the displayed character are compared to the pixels of the background. If the pixels are different, then the final screen pixel is made black. If the pixels are the same, the final screen pixel is made white. Figure 15-15 shows the rule table for the XOR mode.

If you've looked at the example in Figure 15-15, you might be wondering just how useful the XOR mode is, since the text is virtually unreadable against the patterned background. But the XOR mode is very useful when you want to *move* text over a background without affecting it. Take a look at the rule table: We've added a fourth column this time, indicating the results when XOR is called a *second* time. What happens is that the background pattern is *restored.* Consequently, if we CALL TEXTMODE(2), print the text, then print it again at the same spot, the text appears and disappears, leaving the background just as it was. If we then do the same thing at subsequent locations, the result is that the text "ripples" through the background without leaving a messy trail of erased or reversed pixels behind it.

Such an effect can be exploited in any situation where you wish to move text or font symbols across a patterned background. Hopefully springing to the minds of the playful among you is the idea of using XOR to draw Cairo symbols over backgrounds. Figure 15-16 shows a program that takes the train we created earlier with the Cairo font and "moves" it behind a series of Cairo-font palm trees.



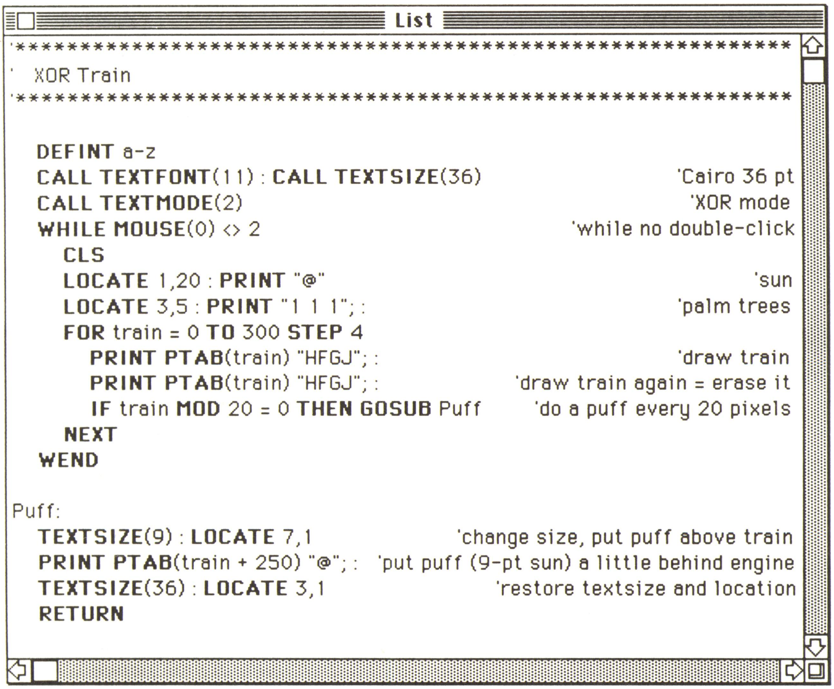
Text Pixel	Background Pixel	Final Pixel	Second XOR (restores background)
○	○	○	○
○	●	●	●
●	○	●	○
●	●	○	●

FIGURE 15-15

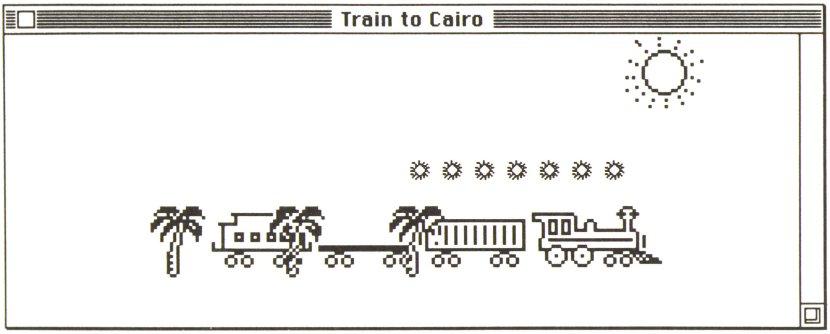
The pixel rules for TEXTMODE(2): The XOR mode

FIGURE 15-16

In XOR text mode, the train is repeatedly drawn and erased as it moves to the right without erasing the palm trees



```
'*************************************************************
'  XOR Train
'*************************************************************


  DEFINT a-z
  CALL TEXTFONT(11) : CALL TEXTSIZE(36)                        'Cairo 36 pt
  CALL TEXTMODE(2)                                                'XOR mode
  WHILE MOUSE(0) <> 2                                 'while no double-click
    CLS
    LOCATE 1,20 : PRINT "@"                                              'sun
    LOCATE 3,5 : PRINT "1 1 1";:                                  'palm trees
    FOR train = 0 TO 300 STEP 4
      PRINT PTAB(train) "HFGJ";:                                  'draw train
      PRINT PTAB(train) "HFGJ";:                   'draw train again = erase it
      IF train MOD 20 = 0 THEN GOSUB Puff           'do a puff every 20 pixels
    NEXT
  WEND

Puff:
  TEXTSIZE(9) : LOCATE 7,1                   'change size, put puff above train
  PRINT PTAB(train + 250) "@";:  'put puff (9-pt sun) a little behind engine
  TEXTSIZE(36) : LOCATE 3,1                    'restore textsize and location
  RETURN
```



TEXTMODE(3): The BIC mode

TEXTMODE(3) is called the *BIC*, or *black is changed*, mode because all of the black pixels in the text characters are changed to white before they are printed on the background. However, all of the white pixels in the characters become, in effect, "transparent" and allow the background pattern to show through. You can see the result in Figure 15-17 on the following page, where the background pattern shows around and in the enclosed areas of the *B*, *I*, and *C*, which were white in the default Copy mode.

FIGURE 15-17

The pixel rules for TEXTMODE(3): The BIC mode

Text Pixel	Background Pixel	Final Pixel
○	○	○
○	●	●
●	○	○
●	●	○

Because the BIC mode changes black text pixels to white and allows the background pattern to show through the text characters' original white pixels, TEXTMODE(3) is obviously not the one to use when we want to print on a white background, since nothing will show at all. But this shortcoming implies the real usefulness of the BIC mode, which is printing labels and other text on dark backgrounds, as Figure 15-17 shows.

THE QUICKDRAW TEXTFACE STATEMENT

You might recall that in Chapter 4, "Introduction to Graphics," we showed you some of the ways you could control the textfaces of printed text with the statement:

CALL TEXTFACE(*n*)

where *n* is a number corresponding to one of the eight specified textfaces (or combinations of specified textfaces). The TEXTFACE values, and their effects when printed, are shown in the table in Figure 15-18.

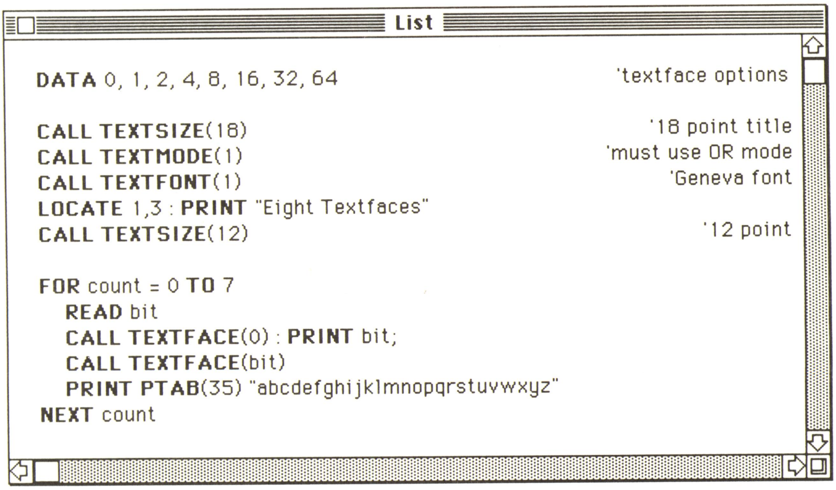
FIGURE 15-18

The TEXTFACE attributes and their values

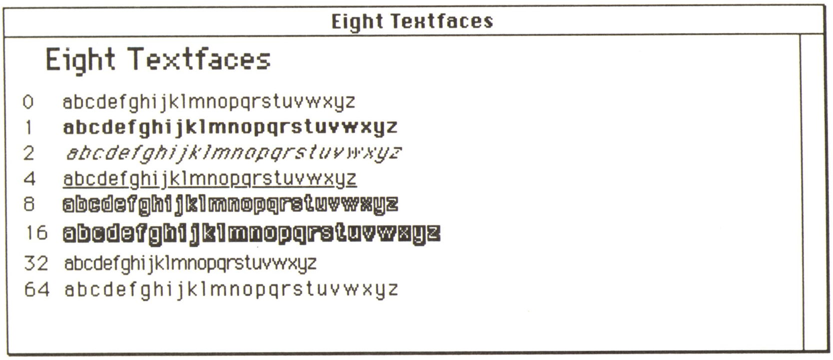
Value	Attribute
0	Plain
1	Bold
2	Italic
4	Underlined
8	Outlined
16	Shadow
32	Condensed
64	Extended

FIGURE 15-19

The eight textfaces



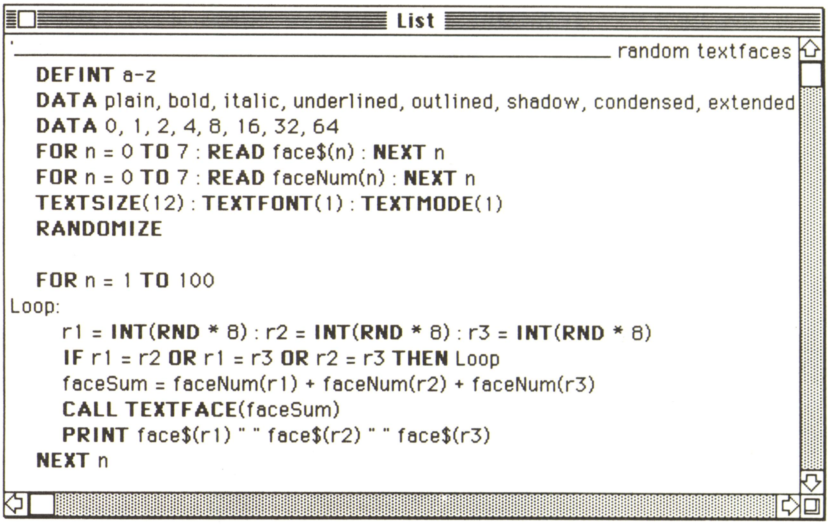
```
DATA 0, 1, 2, 4, 8, 16, 32, 64                    'textface options

CALL TEXTSIZE(18)                                 '18 point title
CALL TEXTMODE(1)                                  'must use OR mode
CALL TEXTFONT(1)                                  'Geneva font
LOCATE 1,3 : PRINT "Eight Textfaces"
CALL TEXTSIZE(12)                                 '12 point

FOR count = 0 TO 7
   READ bit
   CALL TEXTFACE(0) : PRINT bit;
   CALL TEXTFACE(bit)
   PRINT PTAB(35) "abcdefghijklmnopqrstuvwxyz"
NEXT count
```



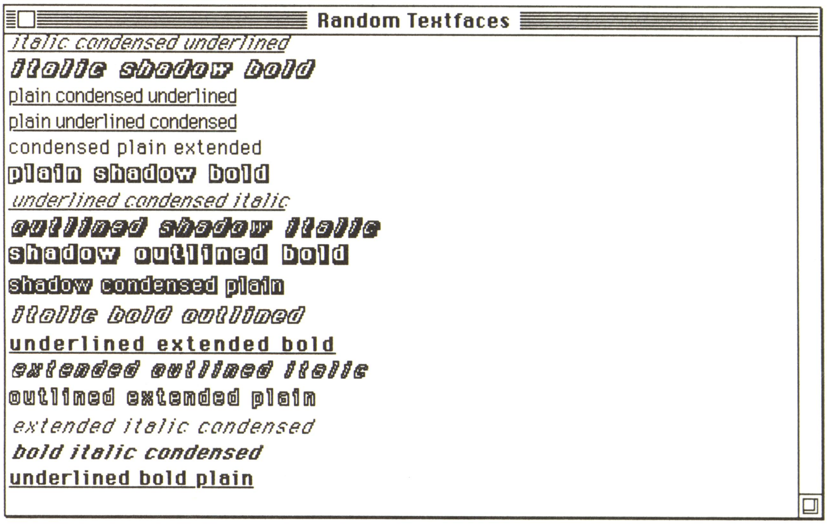
```
  Eight Textfaces
 0   abcdefghijklmnopqrstuvwxyz
 1   abcdefghijklmnopqrstuvwxyz
 2   abcdefghijklmnopqrstuvwxyz
 4   abcdefghijklmnopqrstuvwxyz
 8   abcdefghijklmnopqrstuvwxyz
 16  abcdefghijklmnopqrstuvwxyz
 32  abcdefghijklmnopqrstuvwxyz
 64  abcdefghijklmnopqrstuvwxyz
```

In Chapter 4 we only showed you how to get plain, bold, underlined, and condensed textfaces. But as you can see in Figure 15-18, there are in fact eight specified textfaces available to our BASIC programs. The trick to using the italic, outlined, shadowed, and extended faces is to first call the OR textmode, TEXTMODE(1). The program in Figure 15-19 generates a sampling of all eight of the basic textfaces.

However, we also learned in Chapter 4 that these eight textfaces can be combined by adding their numbers together. So, the statement CALL TEXTFACE(3) would produce bold-italic type because the value 1 calls boldface and the value 2 calls italic. But as with fonts, care must be taken when mixing textfaces because some combinations can create an



```
'_____________________________________________________ random textfaces
  DEFINT a-z
  DATA plain, bold, italic, underlined, outlined, shadow, condensed, extended
  DATA 0, 1, 2, 4, 8, 16, 32, 64
  FOR n = 0 TO 7 : READ face$(n) : NEXT n
  FOR n = 0 TO 7 : READ faceNum(n) : NEXT n
  TEXTSIZE(12) : TEXTFONT(1) : TEXTMODE(1)
  RANDOMIZE

  FOR n = 1 TO 100
Loop:
    r1 = INT(RND * 8) : r2 = INT(RND * 8) : r3 = INT(RND * 8)
    IF r1 = r2 OR r1 = r3 OR r2 = r3 THEN Loop
    faceSum = faceNum(r1) + faceNum(r2) + faceNum(r3)
    CALL TEXTFACE(faceSum)
    PRINT face$(r1) " " face$(r2) " " face$(r3)
  NEXT n
```



```
italic condensed underlined
italic shadow bold
plain condensed underlined
plain underlined condensed
condensed plain extended
plain shadow bold
underlined condensed italic
outlined shadow italic
shadow outlined bold
shadow condensed plain
italic bold outlined
underlined extended bold
extended outlined italic
outlined extended plain
extended italic condensed
bold italic condensed
underlined bold plain
```

FIGURE 15-20

Random textfaces

unattractive "visual disharmony." The program in Figure 15-20 generates random combinations of textfaces, and will give you an idea of what looks good and what doesn't. You can let it run for awhile, then choose Suspend from the Run menu and print a screen dump of the window's content with the Command-Shift-4 key combination.

Our discussion of the four text operations brings us to a QuickDraw routine that affects both text and graphics: MOVETO.

THE QUICKDRAW MOVETO STATEMENT

Earlier in this chapter we created a circle and labeled it *The Circle.* We learned then that we're able to prevent a line of text from obliterating the background pattern by calling TEXTMODE(1), the OR mode. But when we left the example, the label was still tangled up with the left edge of the circle. How can we move the label so that it's nicely centered in the middle of the circle?

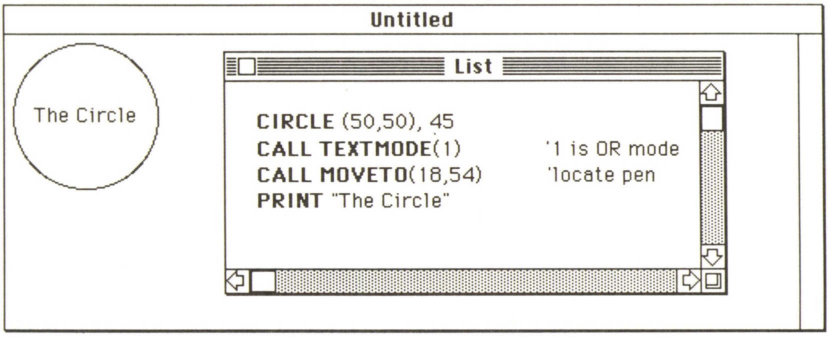
You might immediately think to use the LOCATE statement to move the text to the correct row, and then the PTAB statement to move it horizontally. The problem with this approach is that LOCATE works with text *character* spacing, not *pixel* spacing, which is what we need here. That is, LOCATE moves by *rows,* or a fixed number of pixels all at once, and the number of pixels depends on the current font size. We want to move our label as near to the center of the circle as possible, which does not happen to fall conveniently on a text row. So if we tried a LOCATE 4,1 statement, the label would be printed well below the center of the circle. But if we add the MOVETO statement shown in Figure 15-21, it will be printed in the center as you can see in the Output window.

The MOVETO statement causes the first letter of a text line to be printed at the pixel coordinates specified in the parentheses. The lower left corner of the first letter is referenced in this statement. The general syntax of MOVETO is:

CALL MOVETO(*x*,*y*)

where *x* and *y* are variables representing the horizontal and vertical pixel location where printing is to begin.

In our circle example, we experimented to find the right values to put between the parentheses in the MOVETO statement, and horizontal



```
CIRCLE (50,50), 45
CALL TEXTMODE(1)          '1 is OR mode
CALL MOVETO(18,54)        'locate pen
PRINT "The Circle"
```

FIGURE 15-21

Using the MOVETO statement to put the label in the center of the circle pixel 18 and vertical pixel 54 seemed to print the label as close to the center as we could get it. Unfortunately, there is no magic BASIC statement to "autocenter" a label inside a graphic object like we can easily do with the Align Middle option under the MacPaint Style menu.

It has probably occurred to you that MOVETO works much like PSET, in that it locates an exact *pixel* coordinate in the Output window. But there is much more to MOVETO than that. In fact, it brings us to one of those underlying philosophical concepts of the Macintosh we have repeatedly mentioned. Let's discuss that concept now.

THE QUICKDRAW PEN

QuickDraw borrows its structure from concepts that are often seen in advanced graphics. One of these primary ideas is known as the *QuickDraw pen.* The pen is a graphic metaphor used for drawing lines, shapes, patterns, and text. It has characteristics analogous to an artist's ink pen: a certain *location* on the screen, a certain *size,* a certain *pattern,* and a *mode* (the penmode is not exactly the same as the textmode, as we'll discover). When you first start BASIC, the pen defaults to the same one-pixel-wide line that is used with the PSET, LINE, and CIRCLE statements. Let's see how we draw with the default pen size.

Moving the pen without drawing: MOVETO

When we centered the label in the circle in our earlier example, we used the QuickDraw CALL MOVETO(x,y) function to control where the text began. What MOVETO(x,y) actually controls is the location of the invisible QuickDraw pen. MOVETO does not draw anything on the screen itself, but instead "moves" the pen to the pixel specified by the (x,y) coordinates. The LINE and LINETO statements are used to actually *draw* with the pen, as we'll see in a moment.

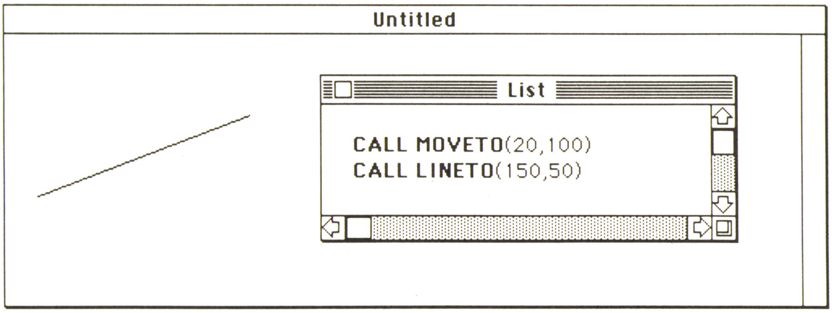
A critical fact to remember is that, in QuickDraw, both text and graphics statements use the same "pen" to locate the starting pixel for a line of text or a graphic image. This is one of the Mac's great features: Positioning text is the same as positioning graphics; both occur on a pixel level.

The LINETO statement

The statement:

CALL LINETO(*x*,*y*)

is provided by QuickDraw to draw a line with the current pen characteristics (size, pattern, and mode) from the pen's current position to the pixel



```
CALL MOVETO(20,100)
CALL LINETO(150,50)
```

FIGURE 15-22

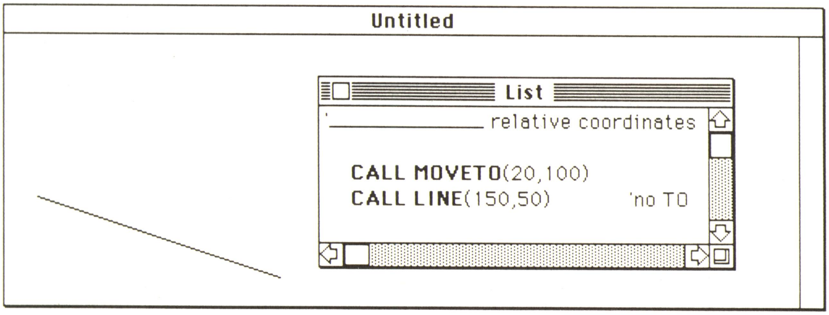
Absolute line drawing with LINETO

coordinate specified by the values x and y in the LINETO statement. Thus, to draw a line from (20,100) to (150,50) would take the two statements shown in Figure 15-22.

Relative MOVE and LINE

The MOVETO and LINETO statements use what are referred to as *absolute coordinates,* which are distinguished from the absolute screen locations discussed in the last chapter. The values inside the parentheses refer to the absolute coordinates of the pixel locations relative to the Output window. QuickDraw provides a second form of MOVETO and LINETO that uses *relative coordinates.* CALL MOVE and CALL LINE work with relative coordinates, as shown in Figure 15-23.

CALL MOVE(*xDelta,yDelta*) moves the pen from its current location to the relative location specified by adding *xDelta* and *yDelta* to the current pen position. Positive values of *xDelta* and *yDelta* move the pen to the right and down and negative values move the pen to the left and up. For example, if the pen is at the coordinates (20,20), we can move it to (10,10) by specifying *CALL MOVE(−10,−10).*



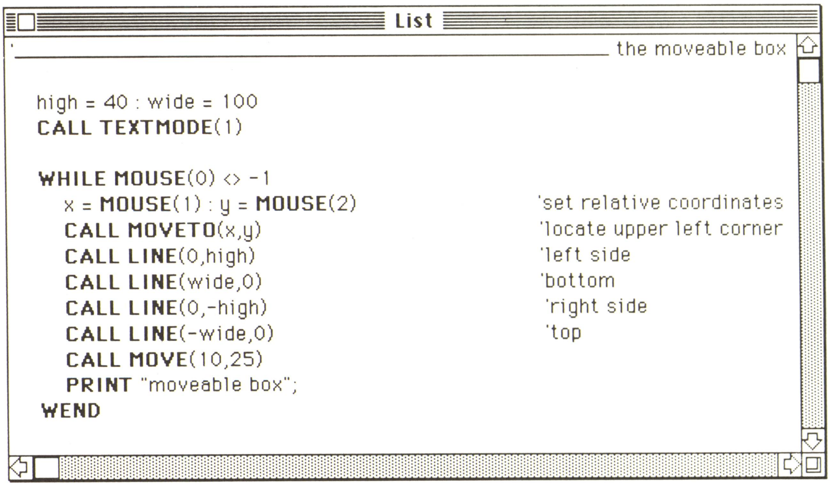
```
'______________ relative coordinates

CALL MOVETO(20,100)
CALL LINE(150,50)          'no TO
```

FIGURE 15-23

Relative line drawing with LINE

In a similar way, the LINE(*xDelta,yDelta*) statement works like LINETO except that *xDelta* and *yDelta* are added to the current coordinates to get the final coordinates. Therefore, in our example in Figure 15-23 the statement *CALL LINE(150,50)* causes the value 150 to be added to the current x coordinate of the pen position (20 + 150), and the value 50 to be added to the y coordinate of the pen position (100 + 50). Note that even though the values for the coordinates in the LINE and LINETO statements are the same in this and the previous program, the programs do very different things.

What good are LINE and MOVE statements with relative coordinates? Their most useful application is when we are drawing an object or printing some text on the screen and we wish to easily move this entire object or text to a new location. If we draw the object using LINE and MOVE statements with relative coordinates, we can reference every line to a single coordinate, like a corner. We can change the position of the corner, and the entire object will be redrawn with no complex calculations. Figure 15-24 shows a program that allows you to move a box with the mouse. Note how simple the LINE statements appear. Imagine how you would accomplish this same effect using absolute coordinates.



```
'_____________________________________________________ the moveable box


high = 40 : wide = 100
CALL TEXTMODE(1)

WHILE MOUSE(0) <> -1
   x = MOUSE(1) : y = MOUSE(2)                  'set relative coordinates
   CALL MOVETO(x,y)                             'locate upper left corner
   CALL LINE(0,high)                            'left side
   CALL LINE(wide,0)                            'bottom
   CALL LINE(0,-high)                            'right side
   CALL LINE(-wide,0)                            'top
   CALL MOVE(10,25)
   PRINT "moveable box";
WEND
```

FIGURE 15-24

Using relative coordinates to make redrawing an object simple

(continued)

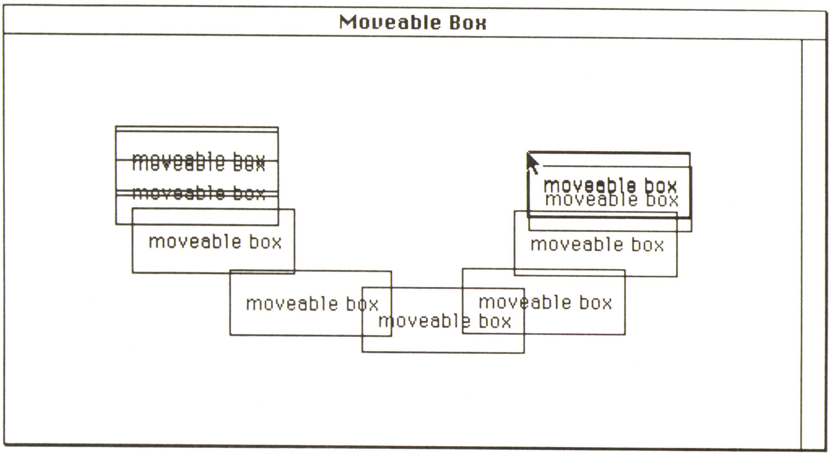

Moveable Box
moveable box
moveable box
moveable box
moveable box
moveable box
moveable box
moveable box
moveable box
moveable box

FIGURE 15-24

(continued)

THE QUICKDRAW PEN SIZE

The QuickDraw pen is used for drawing lines, although it can also be used for plotting. First understand that, as shown in Figure 15-25, the pen is referenced by its upper left corner. This corner is anchored on the "point" that is the mathematical coordinate for the pixel. The QuickDraw pen is actually rectangular and has a variable pixel width and height. When you first start BASIC, the pen defaults to a size of 1 pixel wide by 1 pixel high. The pen always hangs down and to the right of the reference coordinate. If you move the pen to location (9,10), and it has a size of 2 by 3 pixels, the lower right corner of the pen is at (10,12).

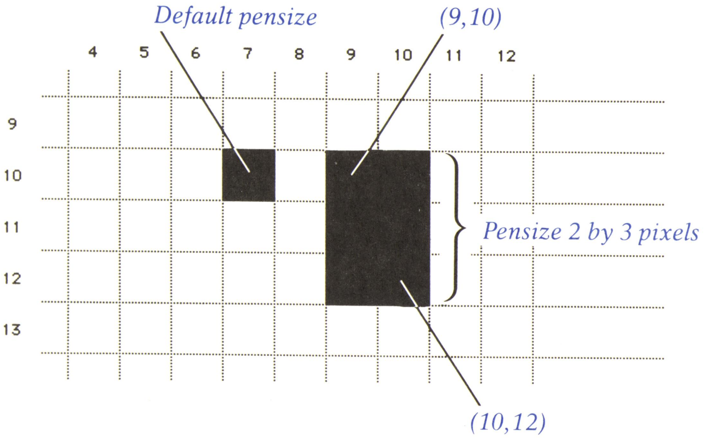
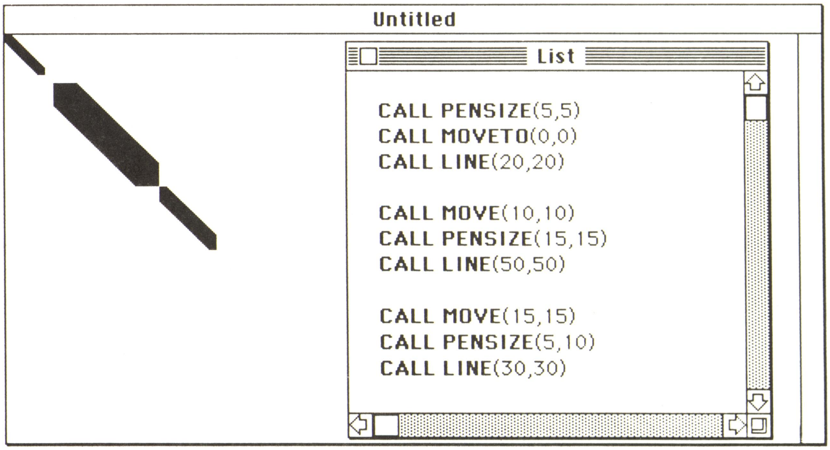

Default pensize
(9,10)
4
5
6
7
8
9
10
11
12
9
10
11
12
13
Pensize 2 by 3 pixels
(10,12)

FIGURE 15-25

The QuickDraw pen hanging down and to the right of its reference coordinates (9,10)

FIGURE 15-26

Using CALL PENSIZE to alter the pen's width and height



```
CALL PENSIZE(5,5)
CALL MOVETO(0,0)
CALL LINE(20,20)

CALL MOVE(10,10)
CALL PENSIZE(15,15)
CALL LINE(50,50)

CALL MOVE(15,15)
CALL PENSIZE(5,10)
CALL LINE(30,30)
```

In case you are wondering if a pen was used when we plotted with the PSET, LINE, and CIRCLE statements—the answer is yes, but that pen was and is not affected by the statements that control the QuickDraw pen. With PSET, LINE, and CIRCLE you are stuck with a pen that is only one pixel wide and one pixel high, and has no patterns and no modes. The QuickDraw pen pattern defaults to black when BASIC is first started, but as we shall see later, it can be given different patterns to draw with.

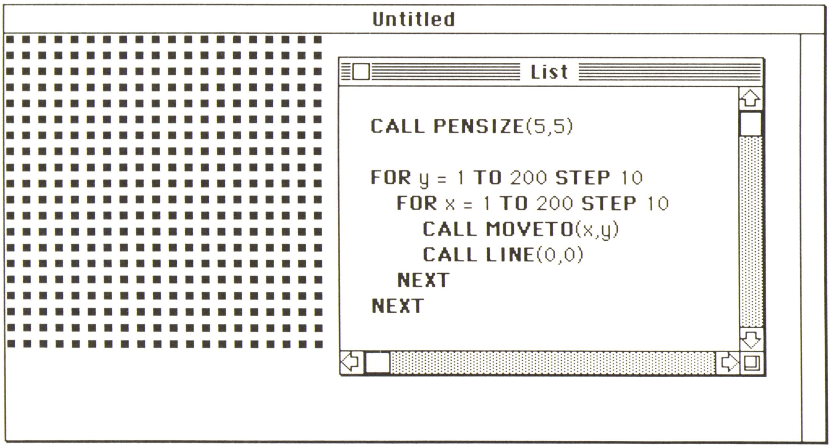
The QuickDraw pen's size can be changed with the statement:

CALL PENSIZE(*width,height*)

Here *width* and *height* are integers that control how high and wide the pen is when it is used with one of the line-drawing statements, as we shall see. Positive values of height and width increase the size of the pen down and to the right of its position. Figure 15-26 shows what happens when we change the pen size to different values. You can imagine that the pen is like a calligrapher's pen, because it leaves a rectangle on the screen.

PLOTTING SINGLE PIXELS WITH THE QUICKDRAW PEN

You might be wondering how a single pixel is plotted with QuickDraw, as we did with PSET. To plot with QuickDraw you need two statements, a MOVE or MOVETO to locate the pen, and a LINE(0,0) statement, which essentially draws a line with no length, which results in a single location being plotted with the current pen size, as shown in Figure 15-27.



```
CALL PENSIZE(5,5)

FOR y = 1 TO 200 STEP 10
   FOR x = 1 TO 200 STEP 10
      CALL MOVETO(x,y)
      CALL LINE(0,0)
   NEXT
NEXT
```

FIGURE 15-27

LINE(0,0) works like PSET and can be used for plotting, but the plot can be any size

QUICKDRAW PARAMETERS

Often *parameters* must be sent to the QuickDraw routine you want to call. In our MOVETO example, *x* and *y* are considered parameters that are "passed" on to the QuickDraw function. To meet QuickDraw requirements, these parameters should be integers. The variables used for parameters with QuickDraw statements can be declared either with the DEFINT statement, or with the % sign following the variable name. These two programs illustrate the two approaches:

```
DEFINT x, y
x = 7 : y = 81
CALL MOVETO(x,y)
```

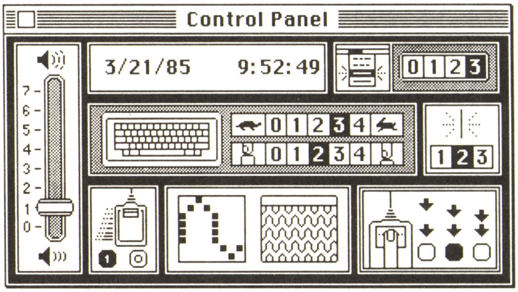
```
x% = 7 : y% = 81
CALL MOVETO(x%,y%)
```

In both cases *x* and *y* are considered integers and accepted by the QuickDraw CALL MOVETO. When the type of parameters for the QuickDraw statements are not specified, Microsoft BASIC automatically converts them to integers.

DRAWING PATTERNS WITH THE QUICKDRAW PEN

We mentioned earlier that besides being able to change its size, the QuickDraw pen can draw with patterns. Patterns are a fundamental aspect of the Macintosh, and especially of QuickDraw. Using patterns requires that we learn a new technique for transferring information between Microsoft BASIC and the QuickDraw routines, but it will be well worth the effort.

If we choose the Control Panel from the Apple menu, we can see the various patterns that Apple has already designed for the background pattern on the Macintosh desktop:



The "pattern display box" in the lower center of the Control Panel lets us choose what pattern is assigned to the desktop. Clicking in the white bar lets us "flip through" the patterns. Every pattern is actually made up of many 8- by 8-pixel blocks. These blocks are laid down on the screen side by side. The panel on the left shows how one of these blocks looks when it is blown up. MacPaint users will recognize this blowup as a *FatBits* representation of the background pattern.

Microsoft BASIC lets us create *any* pattern we want and then affix this pattern to the pen. Then when the pen draws something with the LINE statement or with a QuickDraw drawing routine, it leaves a "trail" of the pattern on the screen, rather than the all-black default pattern we are familiar with. We can copy the patterns created by Apple to the pen, or we can create custom varieties that no one has imagined before. We can also use the patterned pen to "paint" the inside of special shapes, as we shall see later in the chapter.

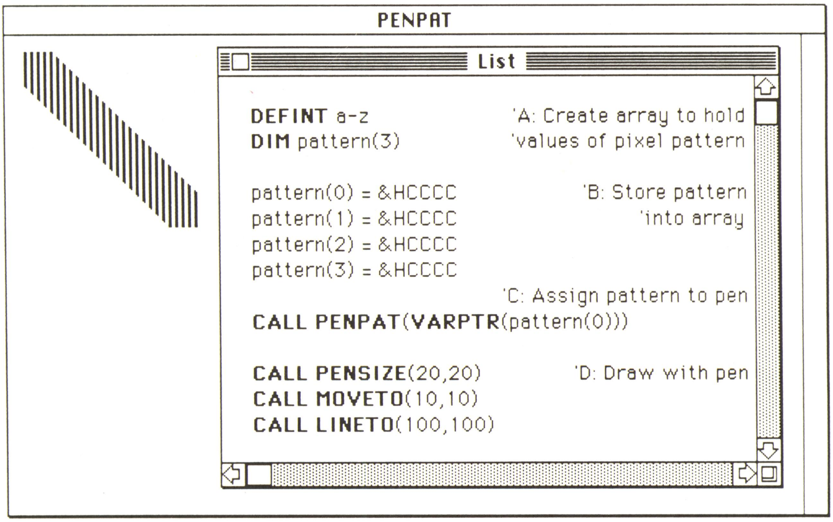
Drawing patterns with the pen

Figure 15-28 shows a program that creates a vertical-stripe pattern for the pen, and then uses the pen to draw a fat 20- by 20-pixel diagonal line with this pattern in the Output window. Let's examine the program briefly to get a general idea of how it works, then examine in more detail how its individual parts work.

Simply put, pen patterns are created by controlling the dots that make up a single 8- by 8-pixel block. These dots are converted to numeric values that must be stored somewhere in the Macintosh's *random access memory,* or RAM. (RAM is different from ROM in that if the power is turned off, the information in RAM is erased.)

FIGURE 15-28

A pen pattern example



```
DEFINT a-z                          'A: Create array to hold
DIM pattern(3)                      'values of pixel pattern

pattern(0) = &HCCCC                   'B: Store pattern
pattern(1) = &HCCCC                       'into array
pattern(2) = &HCCCC
pattern(3) = &HCCCC
                                   'C: Assign pattern to pen
CALL PENPAT(VARPTR(pattern(0)))

CALL PENSIZE(20,20)                   'D: Draw with pen
CALL MOVETO(10,10)
CALL LINETO(100,100)
```

To get the right numeric values into RAM for the patterns we want, BASIC requires that the values be organized in a specific manner in a *four-element integer array.* We will learn how these numbers are organized soon, but for now let's just focus on the way the pen pattern is set up and used. The sections below are keyed to the comments in the program listing in Figure 15-28.

A: Create array to hold values of pixel pattern. The array called *pattern* in the statement *DIM pattern(3)* will hold the values for the dot pattern we want to assign to the pen. The *DEFINT a-z* statement preceding the *pattern* dimension statement defines all variables in the program to be of integer type.

B: Store pattern into array. Next there are four assignment statements that set the four pattern-array elements to specific values. Note that the numbers we are putting into the array are expressed in *hexadecimal* equivalents. Hexadecimal is simply a number expressed in a different numbering system than the standard decimal system we have been using. The hexadecimal value could be specified in decimal if we wanted, but hexadecimal is chosen for reasons that will become clearer later. (*Binary* is another number system we will discuss.)

C: Assign pattern to pen. The *CALL PENPAT* statement is used to assign the values in the pattern array to the current pen pattern. We mentioned that the pen pattern must be stored in the RAM of the Macintosh.

Therefore, the PENPAT call requires that we tell it the *memory address* of the pattern array containing the pen pattern values. Needing to give a memory address to our BASIC program is something we have never come across before, but it is not uncommon.

VARPTR is an advanced Microsoft BASIC function that is used to find out *where* in RAM a particular BASIC variable or array is stored. What VARPTR does is return the *address* of the variable or array stored in RAM. We say that VARPTR is a *pointer* (PTR) to the *variable* (VAR). The VARPTR address information is usually used by programs that need to communicate with some external assembly language program, or which need to directly manipulate an array or a string. Although scorned by many computer-science perfectionists, VARPTR is sometimes the only way certain things can be done. The BASIC manual contains more information about how the different variable types are organized.

When the program is first run, the values in *pattern(0)*, *pattern(1)*, *pattern(2)*, and *pattern(3)* are placed sequentially at some fixed location in memory, starting with *pattern(0)*. Therefore the expression:

```
VARPTR(pattern(0))
```

in the CALL PENPAT statement gives the correct starting address for the PENPAT routine to get the pattern values.

You will see VARPTR used frequently in upcoming QuickDraw functions and statements to return the address of arrays holding different things, such as coordinates.

D: Draw with pen. The pen pattern is now defined as a stripe pattern. The pen size is increased to 20 by 20 pixels by *PENSIZE(20,20)*. Since pattern blocks are only 8 by 8 pixels in size, we can see from the results that the pattern is uniformly distributed across the entire pen thickness.

Try substituting these other hexadecimal numbers for the *CCCCs* in the *pattern* lines in the program and watch the results:

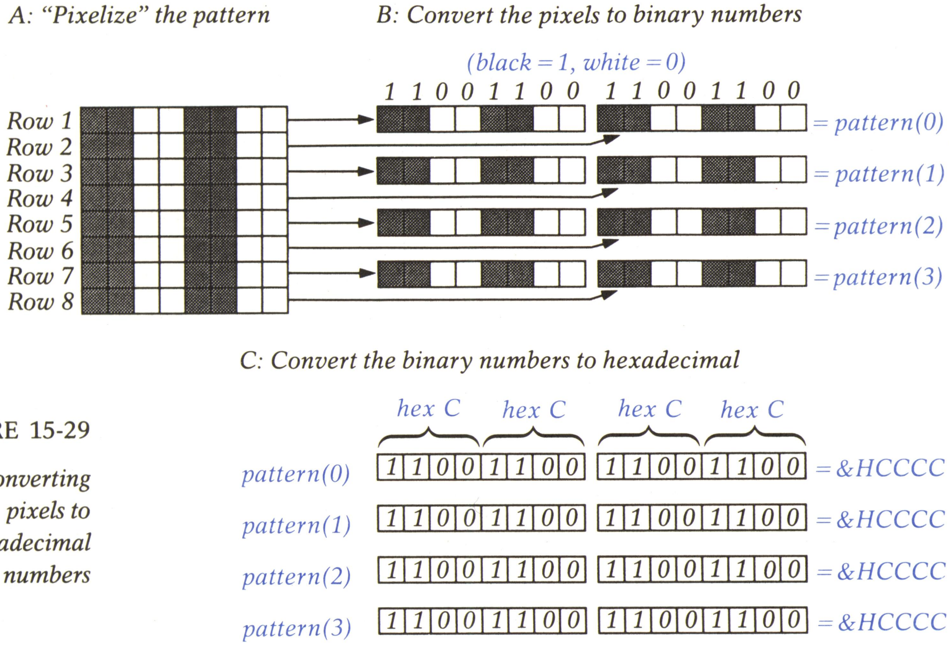
FF80 8080 FF08 0808 (bricks)
F874 2247 8F17 2271 (weave)

(Note that after you enter the numbers for the brick pattern, the first zero in 0808 disappears. Don't be alarmed; BASIC will remove the initial zeros from such numbers.)

Finding the hexadecimal numbers

Now that we have a bird's-eye view of how patterns are assigned to the pen and drawn with QuickDraw statements, you're probably wondering how we came up with those curious &HCCCC hexadecimal numbers in each *pattern* statement. Let's go through the steps that we need to follow to convert patterns to the hex numbers we need. These steps are illustrated in the three sections of Figure 15-29.

A: "Pixelize" the pattern. When we looked at the pattern display box of the Control Panel, we saw that each pattern is constructed of blocks that are eight pixels square. Our first step, then, is to break our pattern down into its eight-pixel-square grid. In this case, our pattern consists of two columns of black pixels, followed by two columns of white pixels, followed by two columns of black, and finally two columns of white.


A: "Pixelize" the pattern

B: Convert the pixels to binary numbers

C: Convert the binary numbers to hexadecimal
(black = 1, white = 0)
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6
Row 7
Row 8
= pattern(0)
= pattern(1)
= pattern(2)
= pattern(3)
hex C
hex C
hex C
hex C
pattern(0)
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
= &HCCCC
pattern(1)
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
= &HCCCC
pattern(2)
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
= &HCCCC
pattern(3)
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
= &HCCCC

FIGURE 15-29

Converting pixels to hexadecimal numbers

B: Convert the pixels to binary numbers. Although our pixel-grid contains eight rows, we saw in our pen-pattern example that BASIC stores the pattern as integers in a four-element array. In BASIC, integers are two bytes long; each byte consists of eight bits, so each integer contains a total of sixteen bits.

Now, each bit corresponds to a pixel of the pattern, and in the binary system each bit can be either a 1 or a 0. Since in the Macintosh pixels are either black or white, a black-pixel bit is assigned the number 1, and a white-pixel bit is assigned 0.

So, our second step involves creating four, 16-bit (2-byte) sections, each corresponding to the integer we need in each of the four *pattern* statements. This is done by "bringing up" the even-numbered rows of the pixel grid and appending them to the odd-numbered rows, then assigning the appropriate binary number to each bit. This process is shown in step B of Figure 15-29.

C: Convert the binary numbers to hexadecimal. The third step is simply a matter of taking each 4-bit, binary section of each pattern integer and looking up its corresponding hexadecimal number in the table shown in Figure 15-30. In our present example, each 4-bit section's binary number for the first integer is 1100; when we look at the table we find that this corresponds to C. So, our hex value for the first pattern line (and the other three, in this example) is CCCC, which we assign to the *pattern(0)* statement. BASIC requires that a hex number be preceded with an *&H* operator, which we included in the *pattern* lines in our pen-pattern example.

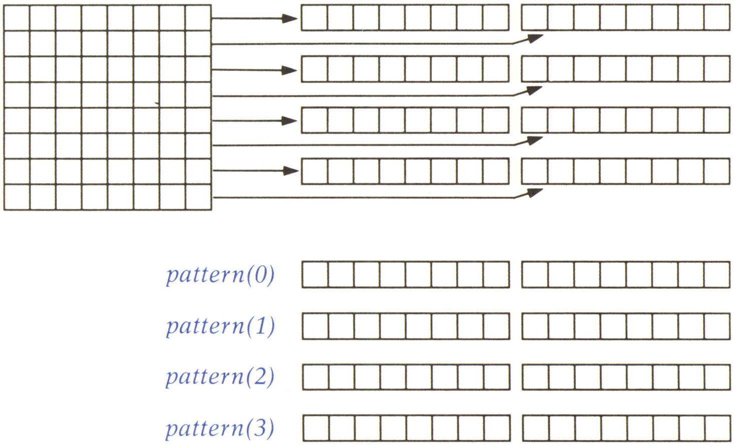
FIGURE 15-30

A table for converting 4-bit binary numbers to hexadecimal

Binary	Hexadecimal	Binary	Hexadecimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

FIGURE 15-31

A blank form for making custom patterns

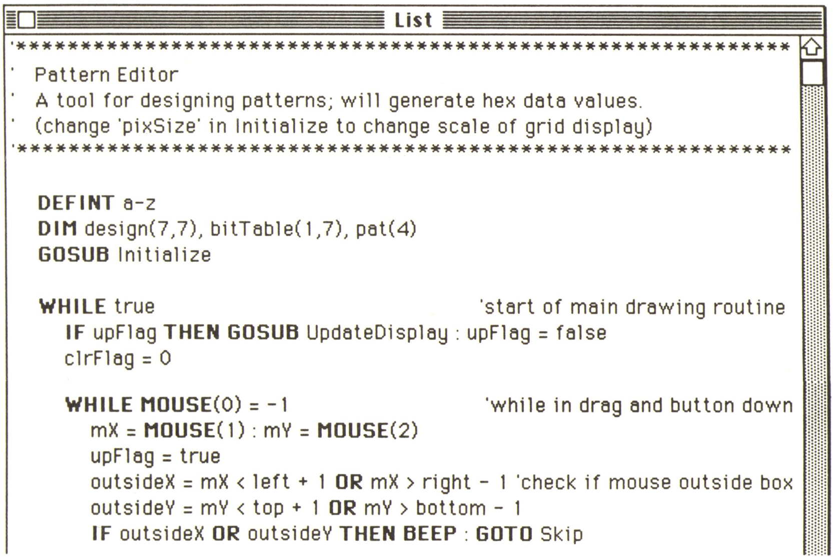


Now that you've learned how to break down patterns and convert them to their hexadecimal equivalents, we have included a blank form in Figure 15-31 that you can use to make up your own patterns and generate the desired hex values.

For those of you who feel inspired by the ability to create patterns, we offer the BASIC program shown in Figure 15-32. The program creates a window with a pair of boxes, similar to those we saw in the pattern display window on the Control Panel. The brick pattern is shown in

FIGURE 15-32

The Pattern-Editor program listing

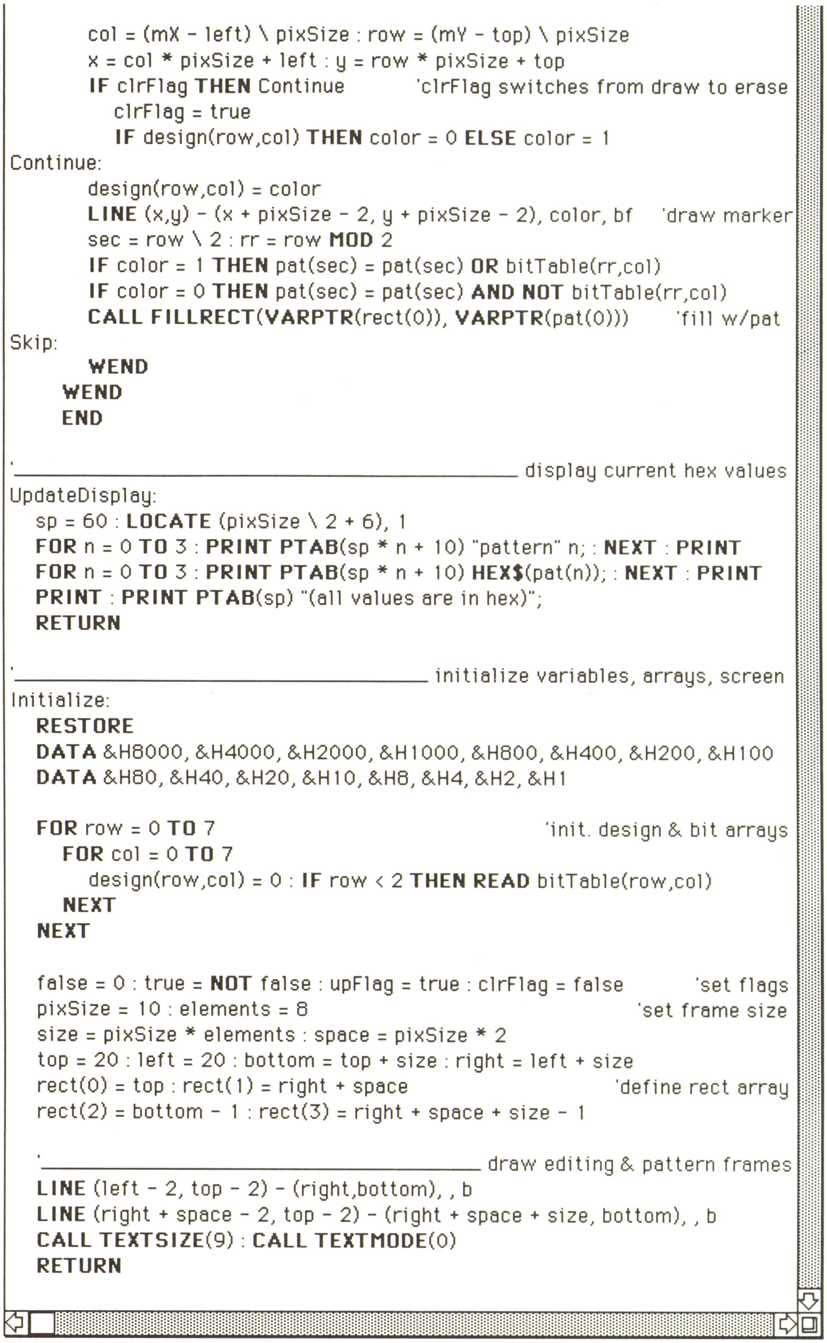


```
'************************************************************
'  Pattern Editor
'  A tool for designing patterns; will generate hex data values.
'  (change 'pixSize' in Initialize to change scale of grid display)
'************************************************************

  DEFINT a-z
  DIM design(7,7), bitTable(1,7), pat(4)
  GOSUB Initialize

  WHILE true                               'start of main drawing routine
    IF upFlag THEN GOSUB UpdateDisplay : upFlag = false
    clrFlag = 0

    WHILE MOUSE(0) = -1                     'while in drag and button down
      mX = MOUSE(1) : mY = MOUSE(2)
      upFlag = true
      outsideX = mX < left + 1 OR mX > right - 1 'check if mouse outside box
      outsideY = mY < top + 1 OR mY > bottom - 1
      IF outsideX OR outsideY THEN BEEP : GOTO Skip
```

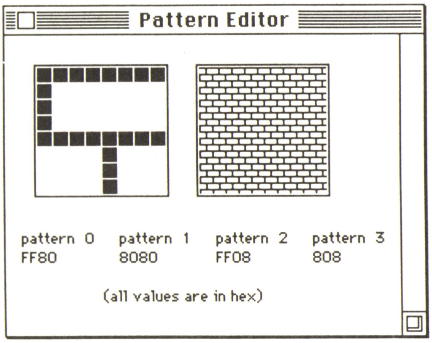
(continued)

FIGURE 15-32

(continued)



```
         col = (mX - left) \ pixSize : row = (mY - top) \ pixSize
         x = col * pixSize + left : y = row * pixSize + top
         IF clrFlag THEN Continue          'clrFlag switches from draw to erase
            clrFlag = true
            IF design(row,col) THEN color = 0 ELSE color = 1
Continue:
         design(row,col) = color
         LINE (x,y) - (x + pixSize - 2, y + pixSize - 2), color, bf    'draw marker
         sec = row \ 2 : rr = row MOD 2
         IF color = 1 THEN pat(sec) = pat(sec) OR bitTable(rr,col)
         IF color = 0 THEN pat(sec) = pat(sec) AND NOT bitTable(rr,col)
         CALL FILLRECT(VARPTR(rect(0)), VARPTR(pat(0)))       'fill w/pat
Skip:
         WEND
      WEND
      END

'________________________________________________ display current hex values
UpdateDisplay:
   sp = 60 : LOCATE (pixSize \ 2 + 6), 1
   FOR n = 0 TO 3 : PRINT PTAB(sp * n + 10) "pattern" n; : NEXT : PRINT
   FOR n = 0 TO 3 : PRINT PTAB(sp * n + 10) HEX$(pat(n)); : NEXT : PRINT
   PRINT : PRINT PTAB(sp) "(all values are in hex)";
   RETURN

'_______________________________________ initialize variables, arrays, screen
Initialize:
   RESTORE
   DATA &H8000, &H4000, &H2000, &H1000, &H800, &H400, &H200, &H100
   DATA &H80, &H40, &H20, &H10, &H8, &H4, &H2, &H1

   FOR row = 0 TO 7                                    'init. design & bit arrays
      FOR col = 0 TO 7
         design(row,col) = 0 : IF row < 2 THEN READ bitTable(row,col)
      NEXT
   NEXT

   false = 0 : true = NOT false : upFlag = true : clrFlag = false        'set flags
   pixSize = 10 : elements = 8                                      'set frame size
   size = pixSize * elements : space = pixSize * 2
   top = 20 : left = 20 : bottom = top + size : right = left + size
   rect(0) = top : rect(1) = right + space                         'define rect array
   rect(2) = bottom - 1 : rect(3) = right + space + size - 1

   '____________________________________________ draw editing & pattern frames
   LINE (left - 2, top - 2) - (right,bottom), , b
   LINE (right + space - 2, top - 2) - (right + space + size, bottom), , b
   CALL TEXTSIZE(9) : CALL TEXTMODE(0)
   RETURN
```



```
pattern 0   pattern 1   pattern 2   pattern 3
FF80        8080        FF08        808

          (all values are in hex)
```

FIGURE 15-33

A brick pattern made with the Pattern-Editor program

Figure 15-33. The box on the left is the 8- by 8-pixel FatBits view of the actual-sized pattern in the right box. You create the pattern in the FatBit box by dragging the pointer to create or erase a series of pixels, or you can click individual pixels to make them black or white.

The magic of these boxes, though, is that they *automatically* convert the pattern into its integer hexadecimal values, which are displayed near the bottom of the window. Once we've created the pattern we want to use, we can simply transfer the hex numbers to our BASIC program and plug them into our pattern statements. So even though we might have to do a bit of typing to enter the program, it could simplify our programming tasks quite a bit later on.

The brick pattern is one that's available in the Control Panel's pattern display window. We can copy other patterns we find there by bringing the Control Panel onto the BASIC desktop, clicking in the white bar to view the different patterns, and then printing the pixel grid for the pattern you want in the Pattern Maker window using Command-Shift-4.

But we're going to make your programming life even easier by revealing the hexadecimal numbers for several of the most interesting and most used patterns, as shown in Figure 15-34. Have fun!

FIGURE 15-34

The hexadecimal values for often-used patterns

Pattern	Values			
Gray desktop	AA55	AA55	AA55	AA55
Light gray	AA00	AA00	AA00	AA00
Dark gray	DD77	DD77	DD77	DD77
Bricks	FF80	8080	FF08	0808
Weave	F874	2247	8F17	2271
Spiral	01FD	85B5	A5BD	81FF

SETTING THE BACKGROUND PATTERN WITH BACKPAT

Although it may not have dawned on you, the white background on which we have been freely spreading our pixels is actually a pattern! It is the pattern made from the data values 0, 0, 0, and 0—supreme white. Assuming that we have previously set a pen pattern with the CALL PENPAT statement, we can change the Output window's background pattern from white to the pen pattern with this statement:

CALL BACKPAT(VARPTR(*pattern*(*0*)))

You should use the QuickDraw BACKPAT routine with some care. If you put too dark a background in the Output window, it may be difficult to see any text or graphics. Since there is no "reset background" QuickDraw call to return the background pattern to white, you must execute a CALL PENPAT statement with white for the pattern and then a CALL BACKPAT statement to set the background to this pattern. The background pattern is activated by using CLS. Another problem with this statement is that it changes the area inside the scroll bar, making it hard to see.

While we are in a critical (protective) mood, you may want to know how to reset the pen to its defaults, should you accidentally set it to some pattern you don't want. The CALL PENNORMAL statement restores the characteristics of the pen to the default setting for size (1 by 1 pixel), pattern (black), and mode (Copy). The pen *location,* however, is not affected by the CALL PENNORMAL statement.

THE QUICKDRAW PENMODE STATEMENT

We learned earlier in this chapter that the textmodes control the way text pixels affect background pixels. In a similar way, QuickDraw offers *pen* modes that let us control how the pixels of the *pen pattern* affect the pixels of the background. QuickDraw has a total of eight penmodes. These modes allow sophisticated relationships between foreground and background pixels. With these modes we can create in our BASIC programs the same pattern effects created with MacPaint's tools, such as the spray can and paint brush, pens of different shapes, and more.

The syntax for setting the penmode is:

CALL PENMODE(*n***)**

where *n* is the *mode number.* The mode numbers are 8 through 15, and each refers to one of the eight pen modes: *Copy, OR, XOR, BIC, Not Copy,*

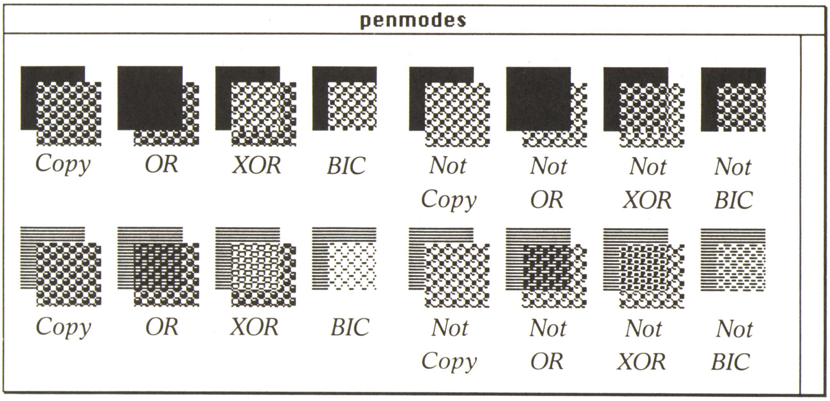


FIGURE 15-35

The eight QuickDraw penmodes on different backgrounds

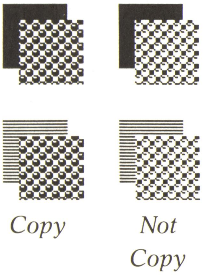
Not OR, Not XOR, and *Not BIC.* Figure 15-35 shows the eight QuickDraw penmodes displayed on a black and white background, and below them the same penmodes on a striped background, giving a more visual insight into how they work.

Now that we have an overview of how the penmodes interact with background patterns, let's look at each one individually.

Copy mode: PENMODE(8), and Not Copy mode: PENMODE(12)

The Copy mode with the pen works just like it does with text: It completely replaces all the pixels of the background with the pen pattern. This is the default mode, which the pen uses automatically until we change the pattern with a CALL PENMODE statement.

As shown in Figure 15-36, the marble pattern drawn with PENMODE(8) wipes out both the black and patterned backgrounds because no background pixels can show through with the Copy mode. PENMODE(12), the Not Copy mode, also completely covers the background pattern but with an *inverted* pattern. That is, Not Copy *reverses* each pixel of the pen pattern before drawing with it: Black pixels become white and white pixels black.



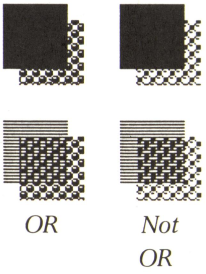
Pen Pixel	Background Pixel	Final Pixel
○	○	○
○	●	○
●	○	●
●	●	●

FIGURE 15-36

The pixel rules for PENMODE(8), Copy, and PENMODE(12), Not Copy

FIGURE 15-37

The pixel rules for PENMODE(9), OR, and PENMODE(13), Not OR



Pen Pixel	Background Pixel	Final Pixel
○	○	○
○	●	●
●	○	●
●	●	●

OR mode: PENMODE(9), and Not OR mode: PENMODE(13)

The OR mode is referred to as the *overlay* mode because the pen pattern is “laid over” the background pattern. With the OR mode, black pixels in the background *override* white pixels in the pen pattern, as shown in the rule table in Figure 15-37. So if the background is *completely* black, the pen pattern disappears on it, as you can see in the figure. But if the background contains white pixels, the OR mode fills in these white pixels with the pen pattern’s black pixels, as shown above in the patterned background display in Figure 15-37. The Not OR mode has the same effect, but the pattern is reversed before it’s overlaid on the background.

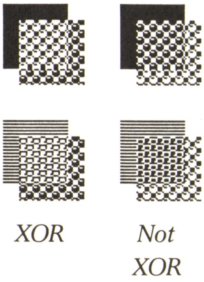
XOR mode: PENMODE(10), and Not XOR mode: PENMODE(14)

The XOR, or *exclusive OR,* mode creates strange effects that nevertheless can be very useful. Recall that XOR’s chief purpose is to allow us to display objects on backgrounds *without permanently erasing or changing the background pattern.* For this reason XOR is useful for programs where text and graphics are *moved* over a background.

The table in Figure 15-38 lists the pixel rules for the XOR mode. The table shows that a black pixel results on the screen only if both the background and pen-pattern pixels are *different;* otherwise the resulting pixel is white. As you can see in the figure, the practical result of using the XOR mode with a black background is that the pen pattern is *inverted:* The black pixels became white and the white pixels became black.

FIGURE 15-38

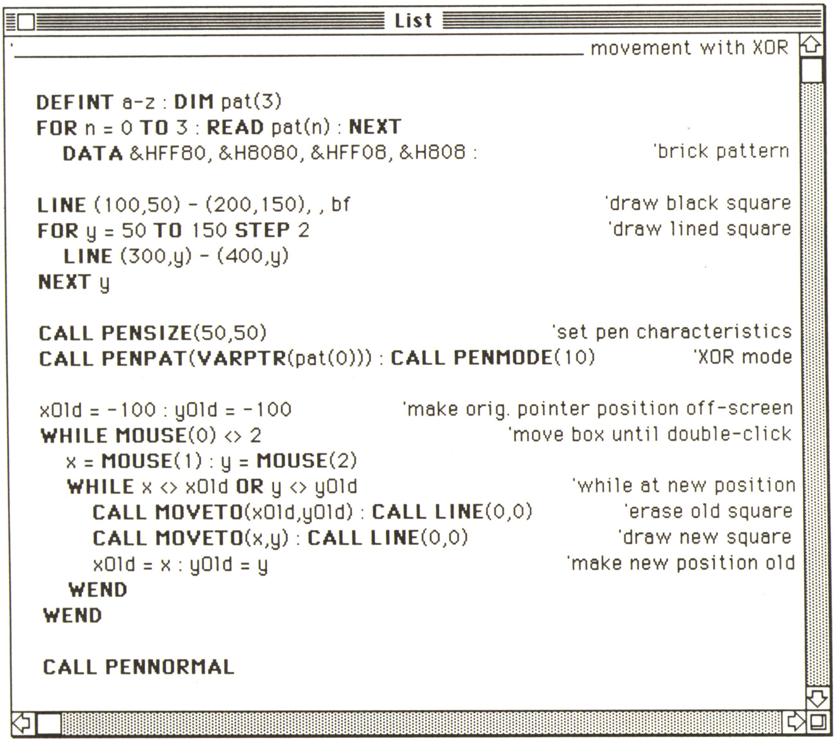
The pixel rules for PENMODE(10), XOR, and PENMODE(14), Not XOR



Pen Pixel	Background Pixel	Final Pixel	Second XOR (restores background)
○	○	○	○
○	●	●	●
●	○	●	○
●	●	○	●

As we saw with the XOR text mode, the chief benefit of the XOR penmode is that it allows graphics to be drawn on background patterns without destroying the pattern. If you draw with a pen pattern in the XOR mode, then redraw the same pen pattern a second time exactly over the first, the original background will be restored because the pixels will be reversed again, back to their original configuration.

The program in Figure 15-39 draws a pen pattern of bricks on the screen based on the coordinates of the mouse. Moving the mouse erases



```
'______________________________________________ movement with XOR

DEFINT a-z : DIM pat(3)
FOR n = 0 TO 3 : READ pat(n) : NEXT
  DATA &HFF80, &H8080, &HFF08, &H808 :                    'brick pattern

LINE (100,50) - (200,150), , bf                          'draw black square
FOR y = 50 TO 150 STEP 2                                 'draw lined square
  LINE (300,y) - (400,y)
NEXT y

CALL PENSIZE(50,50)                                 'set pen characteristics
CALL PENPAT(VARPTR(pat(0))) : CALL PENMODE(10)                   'XOR mode

xOld = -100 : yOld = -100           'make orig. pointer position off-screen
WHILE MOUSE(0) <> 2                            'move box until double-click
  x = MOUSE(1) : y = MOUSE(2)
  WHILE x <> xOld OR y <> yOld                        'while at new position
    CALL MOVETO(xOld,yOld) : CALL LINE(0,0)              'erase old square
    CALL MOVETO(x,y) : CALL LINE(0,0)                     'draw new square
    xOld = x : yOld = y                              'make new position old
  WEND
WEND

CALL PENNORMAL
```

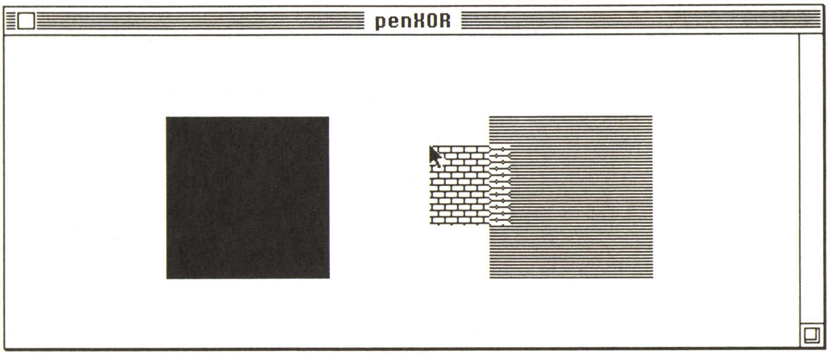
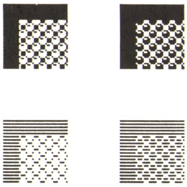


FIGURE 15-39

Using the XOR penmode to move patterns without erasing backgrounds

FIGURE 15-40

Pixel rules for PENMODE(11), BIC



BIC *Not BIC*

Pen Pixel	Background Pixel	Final Pixel
○	○	○
○	●	●
●	○	○
●	●	○

the old pattern and redraws it in the new position. The program only redraws the pattern when the mouse is moved, avoiding eye-tiring flicker from constant updating. The CALL LINE statements do the drawing and erasing. The old and new coordinates of the mouse must be kept track of for this program to work. You can use this program fragment as a skeleton for programs that are to move objects over complex backgrounds.

BIC mode: PENMODE(11), and Not BIC mode: PENMODE(15)

The acronym *BIC* stands for *black is changed,* which is exactly what this mode does: All black pixels of the pen pattern change to white and overlay the pixels on the background pattern. However, the white pixels in the pen pattern are, in effect, *transparent* and allow the black pixels in the background pattern to "show through," as you can see in Figure 15-40. So on all-black backgrounds, the BIC mode inverts pixels like the XOR mode. But since BIC changes all black pixels to white, this mode shows nothing at all on all-white backgrounds.

Now that we know how patterns and modes work, it's time to explore some of the QuickDraw routines for drawing, painting, and filling shapes such as rectangles and ovals.

QUICKDRAW RECTANGLES

QuickDraw allows us to draw rectangles. But if things stopped there, you would be justified in wondering, "Why bother? The BASIC LINE statement works quite well." But QuickDraw allows us to draw a rectangle with any pen pattern, size, and mode we want, and then lets us fill, invert, paint, and erase it.

To draw rectangles on the screen, we use this statement:

CALL FRAMERECT(VARPTR (*rect*(0)))

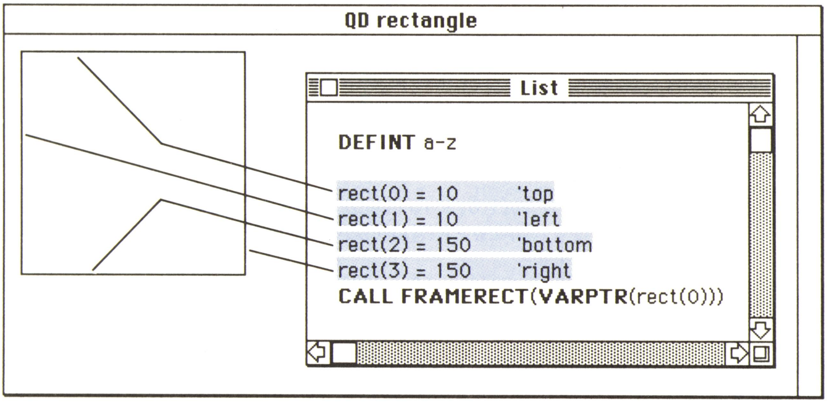
In the world of QuickDraw this is called a *framing* operation. The coordinates of the rectangle must be stored in the integer array *rect* (or whatever name we like) as follows:

```
rect(0) = top
rect(1) = leftSide
rect(2) = bottom
rect(3) = rightSide
```

The VARPTR function in the rectangle-drawing statement works as before when we set the pen pattern: It tells FRAMERECT where to find the absolute coordinates for the rectangle. Figure 15-41 shows an example of FRAMERECT. The DEFINT statement in the example is necessary to set the array to be an integer.

An important distinction here between FRAMERECT and LINE with box option is that FRAMERECT uses the *pen* to frame the rectangle. Therefore, you can assign any pattern and thickness you want to your framed rectangle.

There are four additional QuickDraw functions you can perform on rectangles: *FILLRECT, INVERTRECT, PAINTRECT,* and *ERASERECT.*



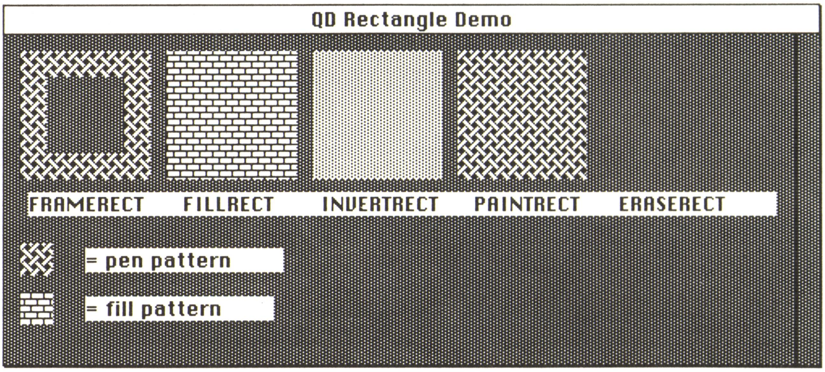
```
DEFINT a-z

rect(0) = 10      'top
rect(1) = 10      'left
rect(2) = 150     'bottom
rect(3) = 150     'right
CALL FRAMERECT(VARPTR(rect(0)))
```

FIGURE 15-41

An example of FRAMERECT

FIGURE 15-42

Framed, filled, inverted, painted, and erased rectangles from the Rectangle-Demo program in Figure 15-43

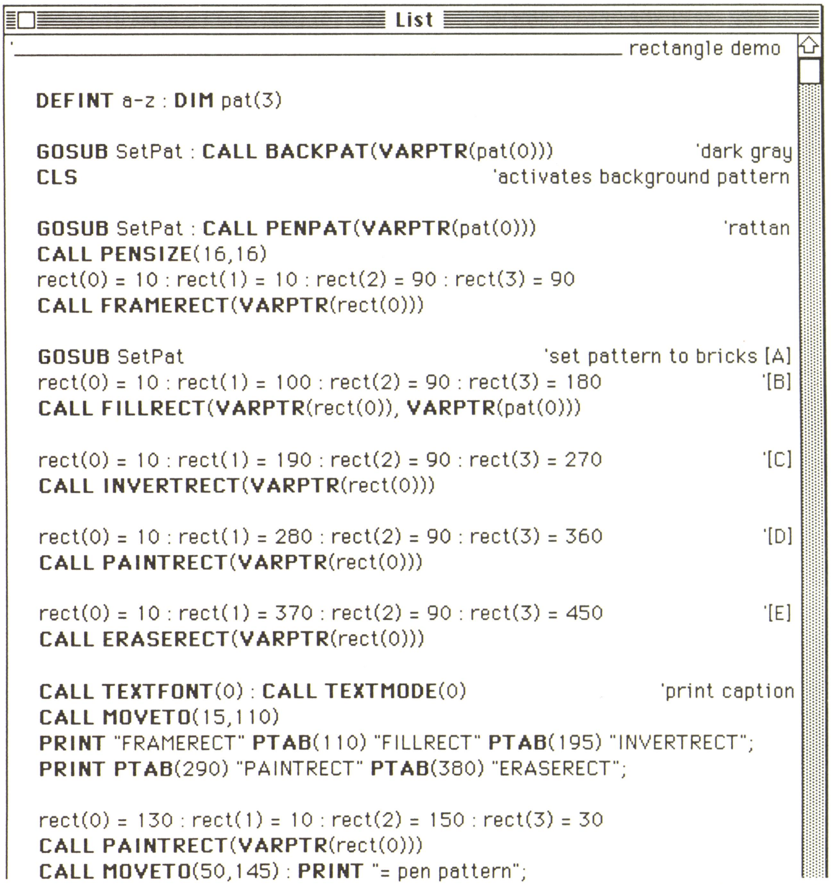


```
FRAMERECT     FILLRECT      INVERTRECT     PAINTRECT     ERASERECT

      = pen pattern

      = fill pattern
```

Figure 15-42 shows the output of a program that demonstrates the five QuickDraw rectangle operations, and Figure 15-43 is the program listing.

FIGURE 15-43

The program that generates output in Figure 15-42

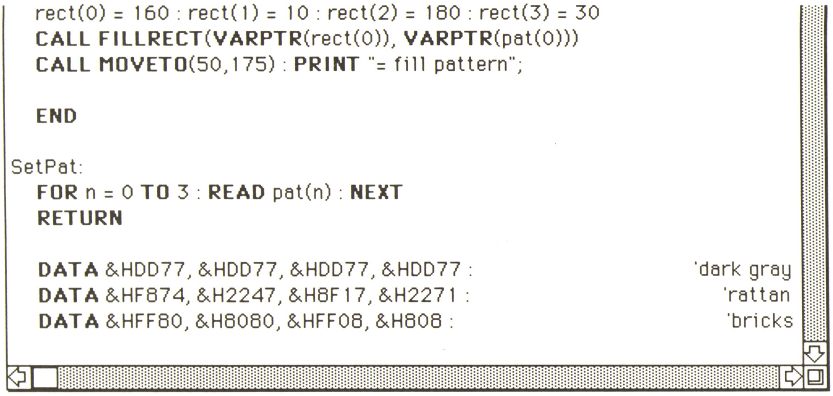


```
'_______________________________________________ rectangle demo

DEFINT a-z : DIM pat(3)

GOSUB SetPat : CALL BACKPAT(VARPTR(pat(0)))                'dark gray
CLS                                          'activates background pattern

GOSUB SetPat : CALL PENPAT(VARPTR(pat(0)))                    'rattan
CALL PENSIZE(16,16)
rect(0) = 10 : rect(1) = 10 : rect(2) = 90 : rect(3) = 90
CALL FRAMERECT(VARPTR(rect(0)))

GOSUB SetPat                                      'set pattern to bricks [A]
rect(0) = 10 : rect(1) = 100 : rect(2) = 90 : rect(3) = 180            '[B]
CALL FILLRECT(VARPTR(rect(0)), VARPTR(pat(0)))

rect(0) = 10 : rect(1) = 190 : rect(2) = 90 : rect(3) = 270            '[C]
CALL INVERTRECT(VARPTR(rect(0)))

rect(0) = 10 : rect(1) = 280 : rect(2) = 90 : rect(3) = 360            '[D]
CALL PAINTRECT(VARPTR(rect(0)))

rect(0) = 10 : rect(1) = 370 : rect(2) = 90 : rect(3) = 450            '[E]
CALL ERASERECT(VARPTR(rect(0)))

CALL TEXTFONT(0) : CALL TEXTMODE(0)                         'print caption
CALL MOVETO(15,110)
PRINT "FRAMERECT" PTAB(110) "FILLRECT" PTAB(195) "INVERTRECT";
PRINT PTAB(290) "PAINTRECT" PTAB(380) "ERASERECT";

rect(0) = 130 : rect(1) = 10 : rect(2) = 150 : rect(3) = 30
CALL PAINTRECT(VARPTR(rect(0)))
CALL MOVETO(50,145) : PRINT "= pen pattern";
```

(continued)

FIGURE 15-43

(continued)



```
   rect(0) = 160 : rect(1) = 10 : rect(2) = 180 : rect(3) = 30
   CALL FILLRECT(VARPTR(rect(0)), VARPTR(pat(0)))
   CALL MOVETO(50,175) : PRINT "= fill pattern";

   END

SetPat:
   FOR n = 0 TO 3 : READ pat(n) : NEXT
   RETURN

   DATA &HDD77, &HDD77, &HDD77, &HDD77 :                 'dark gray
   DATA &HF874, &H2247, &H8F17, &H2271 :                    'rattan
   DATA &HFF80, &H8080, &HFF08, &H808 :                     'bricks
```

Let's see how each of these rectangle operations works, using the labeled sections of the Rectangle-Demo program as examples.

THE FILLRECT STATEMENT

QuickDraw allows a second pattern, called the *fill pattern,* to be used. The fill pattern is created exactly like the pen pattern, as shown in section A of Figure 15-42 where we set the pattern to bricks. Assuming the fill pattern is in the array *pat,* the statement in section B:

```
CALL FILLRECT(VARPTR(rect(0)), VARPTR(pat(0)))
```

draws a filled rectangle at the rectangle coordinates specified by the array *rect,* and the borders of the rectangle are drawn with the current pen pattern and size. Note that this fill is not a true fill, like the one found in MacPaint. The filling takes place within the *entire* rectangle defined in the statement. If you try to fill a same-size rectangle, drawn with a thick pen, the pattern will overlay the original rectangle, pen and all. A "true" fill would fill up to the inner edge of the pen pattern, not obliterate it.

THE INVERTRECT STATEMENT

The INVERTRECT statement in section C takes the current *background* pattern that's within the frame defined by the coordinates in the array *rect,* and inverts every bit in it: turning black bits to white and white bits to black. The syntax is:

```
CALL INVERTRECT(VARPTR(rect(0)))
```

As you can see in Figure 15-43, this turns our dark-gray background to a light-gray background because there are originally more black dots than white ones and the ratio changes to more white than black.

THE PAINTRECT STATEMENT

As shown in section D of the Rectangle-Demo program, you can fill the inside of a rectangle with the pen pattern using the PAINTRECT statement with this syntax:

```
CALL PAINTRECT(VARPTR(rect(0)))
```

Besides giving you a second pattern to paint the interior of the rectangle with, because PAINTRECT uses the pen pattern, all of the eight penmodes are available. FILLRECT does not allow us to use the different penmodes.

THE ERASERECT STATEMENT

The ERASERECT QuickDraw call in section E fills the shape with the current background pattern. This effectively erases the rectangle. The general syntax is:

CALL ERASERECT(VARPTR(_rect_**(0)))**

You can use this QuickDraw routine when you have drawn a rectangle in one pattern and wish to remove it from the screen.

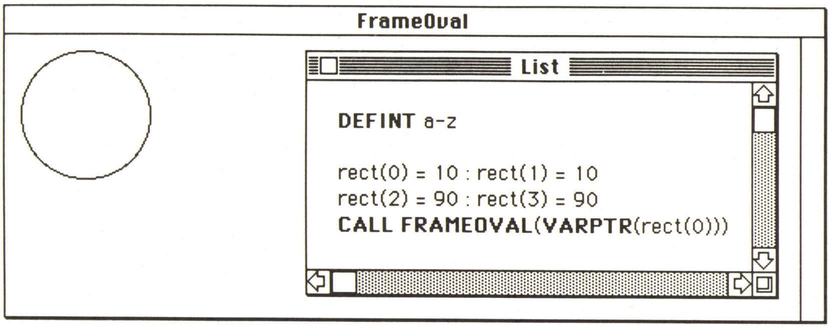
DRAWING QUICKDRAW OVALS AND ROUND-CORNERED RECTANGLES

Another shape that QuickDraw allows us to draw is ovals. The syntax for framing an oval is:

CALL FRAMEOVAL(VARPTR(_rect_**(0)))**

As shown in Figure 15-44, the array *rect* forms the boundary for the edges of the oval. In our example, where the rectangle is a square, a circle is formed. Changing the ratio of the sides of the invisible rectangle alters the shape of the oval.

The four other operations we performed on rectangles can also be performed on ovals: filling, inverting, painting, and erasing. The syntax for each of these operations is:



```
DEFINT a-z

rect(0) = 10 : rect(1) = 10
rect(2) = 90 : rect(3) = 90
CALL FRAMEOVAL(VARPTR(rect(0)))
```

FIGURE 15-44

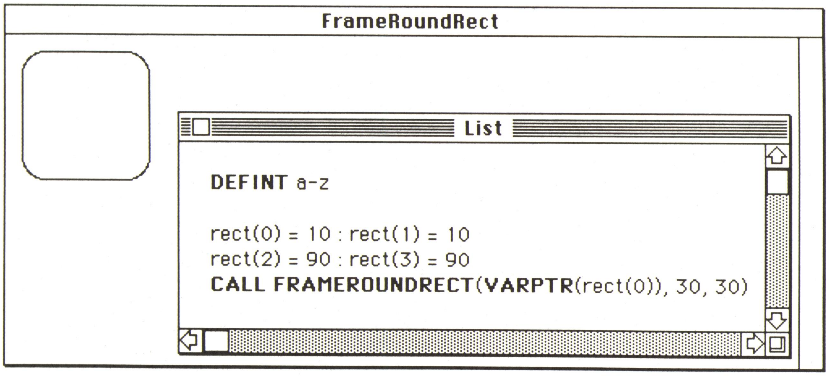
A circle drawn using FRAMEOVAL

CALL FILLOVAL (VARPTR(*rect*(0))), **VARPTR(***pattern*(0)))
CALL INVERTOVAL (VARPTR(*rect*(0)))
CALL PAINTOVAL (VARPTR(*rect*(0)))
CALL ERASEOVAL (VARPTR(*rect*(0)))

There is another QuickDraw routine that produces rectangles with rounded corners, as shown in Figure 15-45. The syntax for drawing a round-cornered rectangle is:

CALL FRAMEROUNDRECT(VARPTR(*rect*(0)), *ovalWidth, ovalHeight*)

The array *rect* holds the top, left, bottom, and right side coordinates for the rectangle. The "roundness" of the corners is set by the values *ovalWidth* and *ovalHeight.* These, as shown in Figure 15-45, represent the horizontal and vertical diameters (in pixels) of a circle that forms the arc for the rounded corner. If the values are the same, a circular corner is drawn; if the values are different, the corners become oval-shaped.



```
DEFINT a-z

rect(0) = 10 : rect(1) = 10
rect(2) = 90 : rect(3) = 90
CALL FRAMEROUNDRECT(VARPTR(rect(0)), 30, 30)
```

FIGURE 15-45

A round-cornered rectangle drawn by FRAMEROUNDRECT

As with the rectangle and oval QuickDraw options, the fill, invert, paint, and erase operations can be performed on round-cornered rectangles with the statements:

CALL FILLROUNDRECT (VARPTR(*rect*(0)), *ovalWidth*, *ovalHeight*, **VARPTR(***pattern*(0)))
CALL INVERTROUNDRECT (VARPTR(*rect*(0)), *ovalWidth*, *ovalHeight*)
CALL PAINTROUNDRECT (VARPTR(*rect*(0)), *ovalWidth*, *ovalHeight*)
CALL ERASEROUNDRECT (VARPTR(*rect*(0)), *ovalWidth*, *ovalHeight*)

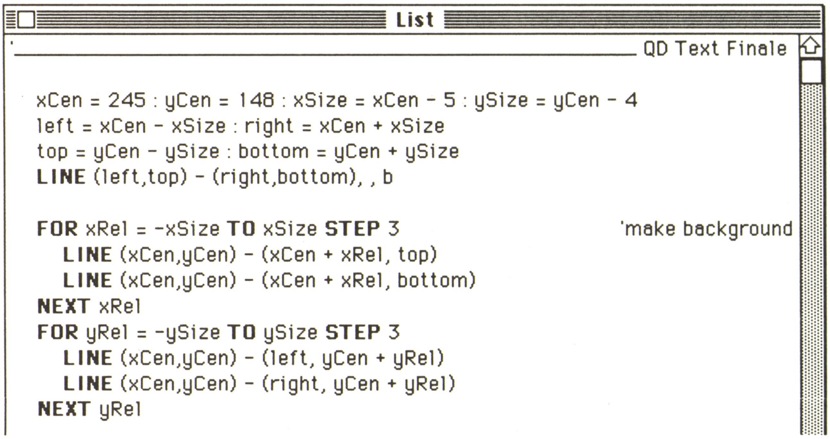
REFRESH-YOURSELF WINDOW

Congratulations are now in order: You have dealt with an enormous amount of complex information in this chapter. But the result is that now you have the knowledge to control the visible world you want to portray on your Macintosh screen. We have learned how to use the QuickDraw calls for controlling text and graphics down to the very pixel. Much more could be said about QuickDraw, but you're well-equipped to set off on your own explorations of the fantastic capabilities of these wonderful tools.

As a final treat, we've created a program that brings together many of the QuickDraw routines we've learned in this chapter. The program, shown in Figure 15-46, combines the OR text mode with the outline textface and Chicago font to produce characters that can be read on almost any background.

FIGURE 15-46

A program combining QuickDraw text and graphics operations

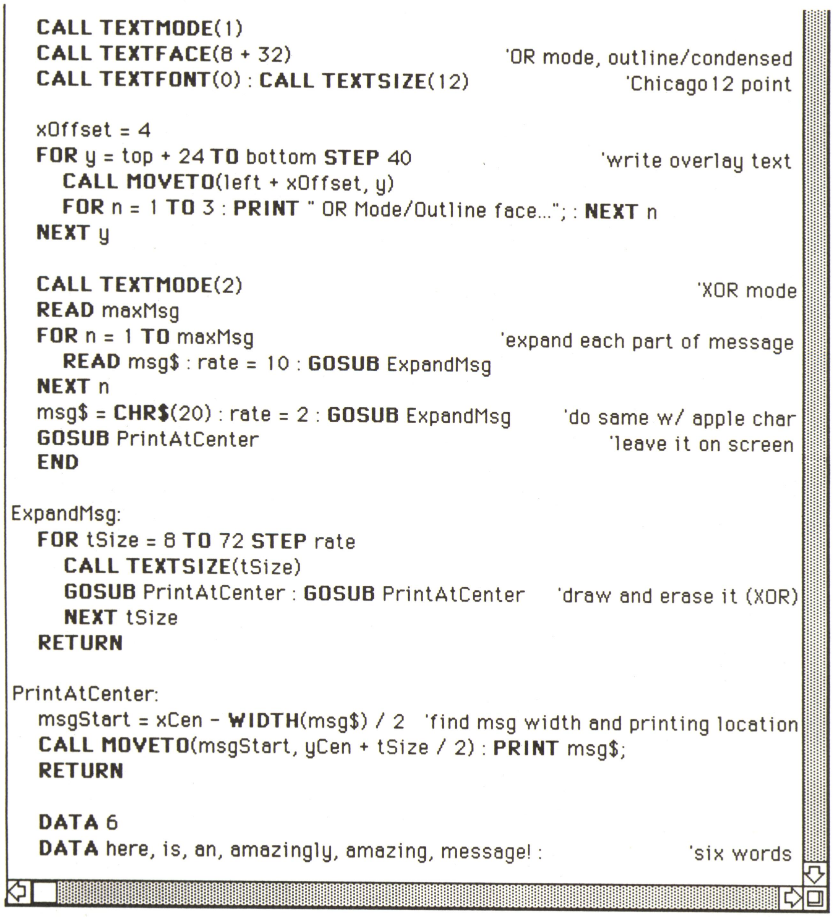


```
'______________________________________________________________ QD Text Finale

xCen = 245 : yCen = 148 : xSize = xCen - 5 : ySize = yCen - 4
left = xCen - xSize : right = xCen + xSize
top = yCen - ySize : bottom = yCen + ySize
LINE (left,top) - (right,bottom), , b

FOR xRel = -xSize TO xSize STEP 3                          'make background
   LINE (xCen,yCen) - (xCen + xRel, top)
   LINE (xCen,yCen) - (xCen + xRel, bottom)
NEXT xRel
FOR yRel = -ySize TO ySize STEP 3
   LINE (xCen,yCen) - (left, yCen + yRel)
   LINE (xCen,yCen) - (right, yCen + yRel)
NEXT yRel
```

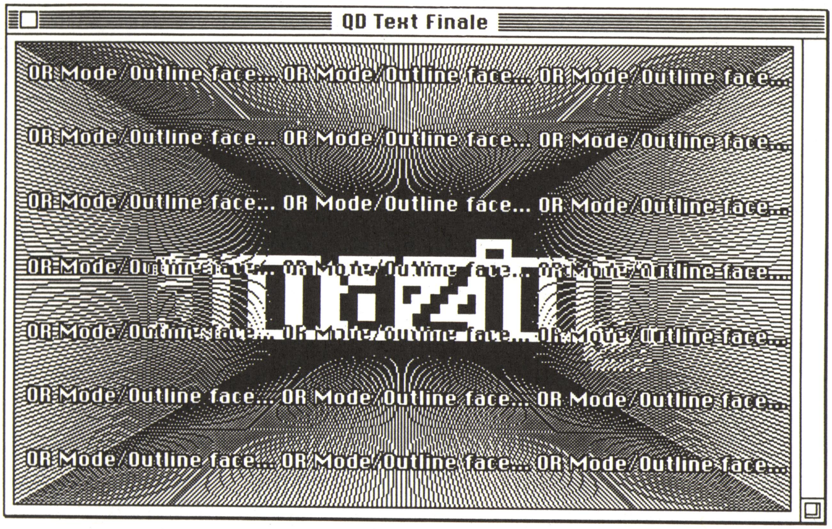
(continued)

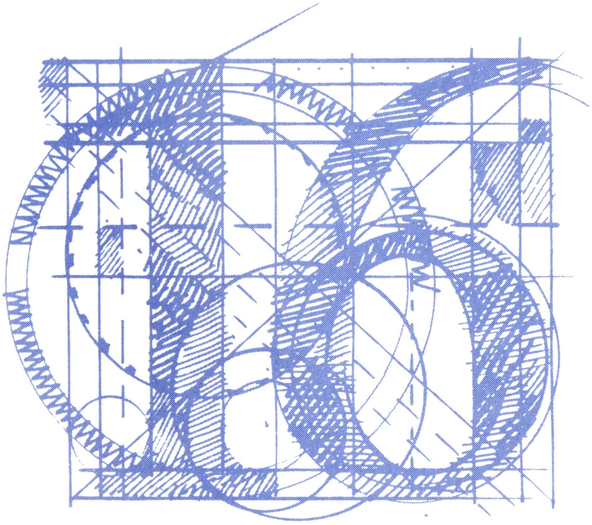
FIGURE 15-46

(continued)



```
   CALL TEXTMODE(1)
   CALL TEXTFACE(8 + 32)                          'OR mode, outline/condensed
   CALL TEXTFONT(0) : CALL TEXTSIZE(12)                       'Chicago12 point

   xOffset = 4
   FOR y = top + 24 TO bottom STEP 40                          'write overlay text
      CALL MOVETO(left + xOffset, y)
      FOR n = 1 TO 3 : PRINT " OR Mode/Outline face...";  : NEXT n
   NEXT y

   CALL TEXTMODE(2)                                                  'XOR mode
   READ maxMsg
   FOR n = 1 TO maxMsg                           'expand each part of message
      READ msg$ : rate = 10 : GOSUB ExpandMsg
   NEXT n
   msg$ = CHR$(20) : rate = 2 : GOSUB ExpandMsg         'do same w/ apple char
   GOSUB PrintAtCenter                                   'leave it on screen
   END

ExpandMsg:
   FOR tSize = 8 TO 72 STEP rate
      CALL TEXTSIZE(tSize)
      GOSUB PrintAtCenter : GOSUB PrintAtCenter     'draw and erase it (XOR)
      NEXT tSize
   RETURN

PrintAtCenter:
   msgStart = xCen - WIDTH(msg$) / 2   'find msg width and printing location
   CALL MOVETO(msgStart, yCen + tSize / 2) : PRINT msg$;
   RETURN

   DATA 6
   DATA here, is, an, amazingly, amazing, message! :                'six words
```


QD Text Finale
OR Mode/Outline face... OR Mode/Outline face... OR Mode/Outline face...



Files

In this chapter we're going to learn about one of the most powerful capabilities of Microsoft BASIC for the Macintosh: files. *This is a complex topic, but we are going to make it as easy as possible for you to understand how you can use files in your programs.*

You're probably asking yourself, "Hmmm. When will I ever need to use files in my programs?" That's a good question. After all, we've made it through the preceding chapters about BASIC without needing files, so why should we need them now? There are three major situations when files are necessary:

1. Your program uses a large set of data.
2. You want to save the output of your program.
3. You want one program to work with many different sets of data.

None of the example programs we've used so far has required a large set of data. These haven't been "typical" programs, though, but instead were examples deliberately kept short and simple. And whenever we've needed data to make our programs work, we obtained it either from the keyboard or from DATA statements built into the program. But in the typical world of programming, almost every program (especially business programs) processes lots of data—too much to be contained in DATA statements or typed in by a user. As we'll see, files can contain large amounts of data that are readily available to be processed.

Typical programs also commonly require that the output be saved for later use as input to the same program, or even to different ones. For example, in Chapter 9, "Arrays and Read-Data," we developed some example programs around the Brothers Four Corporation that used DATA statements to contain the employee names, the number of hours they worked, and the hourly rate. This information is trapped inside the particular program that contains the DATA statements, and is not available for use by any other program. By storing this information in a file, we can make it available to any program that needs it.

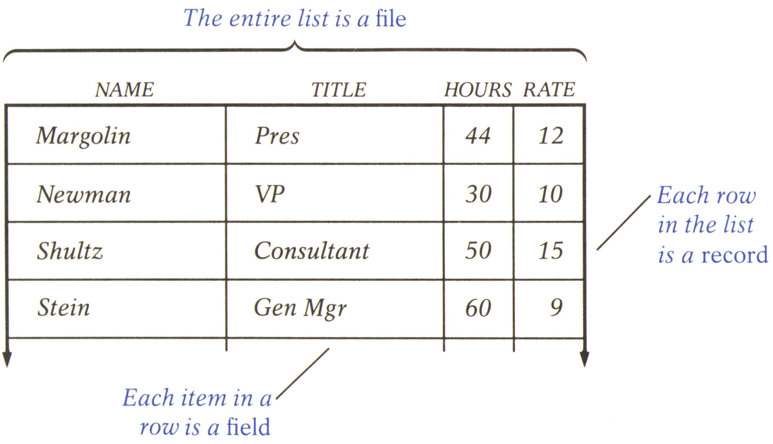
With these two objectives in mind, let's begin our exploration of files and their use in our BASIC programs.

WHAT IS A FILE?

The word *file* can be defined in quite abstract terms, and can take on several different meanings in the world of programming. But a file is usually conceived as a *list* of data stored on disk. The list shown in Figure 16-1 is an example of what we mean. This is a list of the employees of the Brothers Four Corporation, as developed earlier in Chapter 9. The list contains one

FIGURE 16-1

A file arranged like a list



The entire list is a file

NAME	TITLE	HOURS	RATE
Margolin	*Pres*	*44*	*12*
Newman	*VP*	*30*	*10*
Shultz	*Consultant*	*50*	*15*
Stein	*Gen Mgr*	*60*	*9*

Each row in the list is a record

Each item in a row is a field

row for each employee. Each row consists of four columns that contain different information about each employee. You can visualize a file as follows:

- The entire list is a *file.*
- Each row on the list is a *record* in the file.
- Each item on a row is a *field* in the record.

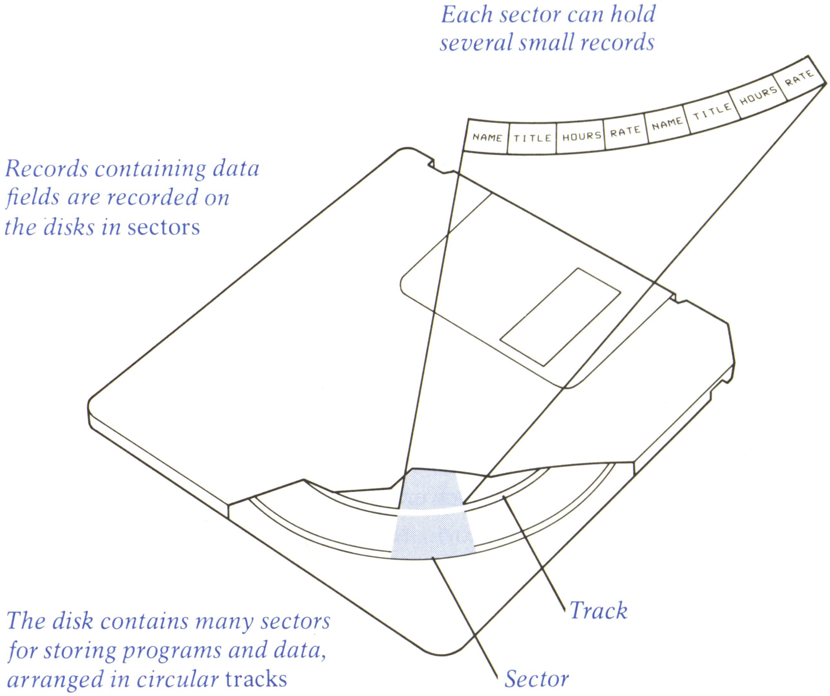
The example file we'll use in this chapter contains records of the payroll information for each of the employees. Each record contains four fields: employee name, title, hours worked, and hourly rate. There are other arrangements possible, but this one will allow us to study all the BASIC statements and functions that are used in conjunction with files.

At this point, it might be helpful to know a little bit about what goes on at the hardware level when working with files. Disk files that are used by BASIC programs are stored on the disk as shown in Figure 16-2 on page 392. The information is "recorded" on the disk in circular *tracks.* When the disk is revolving, information recorded in the tracks can be "read" by a magnetic head inside the Macintosh. Each track is further subdivided into *sectors* (which are bounded by imaginary lines drawn outward from the center of the disk). Each sector contains enough space to store many bytes of information. When there is only a small amount of information in each record, several records may be stored in each sector.

Although we're only going to study the disk files we can create and use in our BASIC programs, we should mention that the *Clipboard* is actually a special disk file that is used by the Macintosh system. Whenever we use the Edit menu to Cut, Copy, or Paste, we're actually using this special disk file without even knowing it.

FIGURE 16-2

The anatomy of a disk file


Each sector can hold several small records
NAME
TITLE
HOURS
RATE
NAME
TITLE
HOURS
RATE
Records containing data fields are recorded on the disks in sectors
The disk contains many sectors for storing programs and data, arranged in circular tracks
Track
Sector

The file concept can also be applied to other methods of handling information that do not use a disk. For example, the *printer* can be considered a *sequential* file, which is used for output operations only. The serial communications port of the Macintosh can be used in BASIC programs as though it were a sequential file, allowing both input and output operations. But our main concern in this chapter is with disk files, so let's get started.

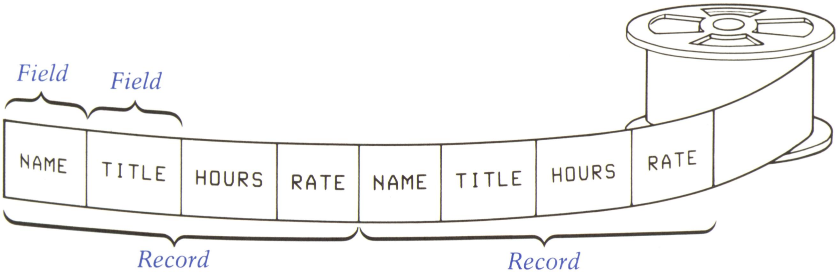
Sequential- versus random-access files

Now that we have a general idea of what a file is, we need to understand how BASIC *accesses* the information stored in a file on disk. There are two methods available: *sequential-access* and *random-access.* We'll be learning about both of these access methods in this chapter, but first, let's just get a general feeling for what these terms mean.

Sequential-access files are the easiest to use of the two types. That is, fewer BASIC statements are required to handle sequential files. The drawback with them is that the information in a sequential file can only be accessed *serially,* that is, one record at a time, in *sequence* from the beginning to the end of the file. Think of BASIC's sequential files as though the data is written on a reel of paper tape, as shown in Figure 16-3.

FIGURE 16-3

BASIC's sequential files are stored and accessed serially, as if written on a reel of paper tape


Field
Field
NAME
TITLE
HOURS
RATE
NAME
TITLE
HOURS
RATE
Record
Record

Applying this idea to our employee file, we would find the first item of data in the file to be the name of the first employee. This would be followed by the employee's title, hours worked, and hourly rate. Then we would find the name of the next employee in the file, and so on. You can see why this kind of file is called sequential: If our program were looking for information pertaining to a particular employee, it would have to look through all the data from the beginning of the tape until it finds the record. This would be like trying to find a particular frame on a videotape, or a particular song on an audio cassette.

Random-access files can provide a faster method of finding particular information in certain circumstances. As we can see in Figure 16-4, a random-access file is like one of those pleated cardboard folders known as an "accordion file."

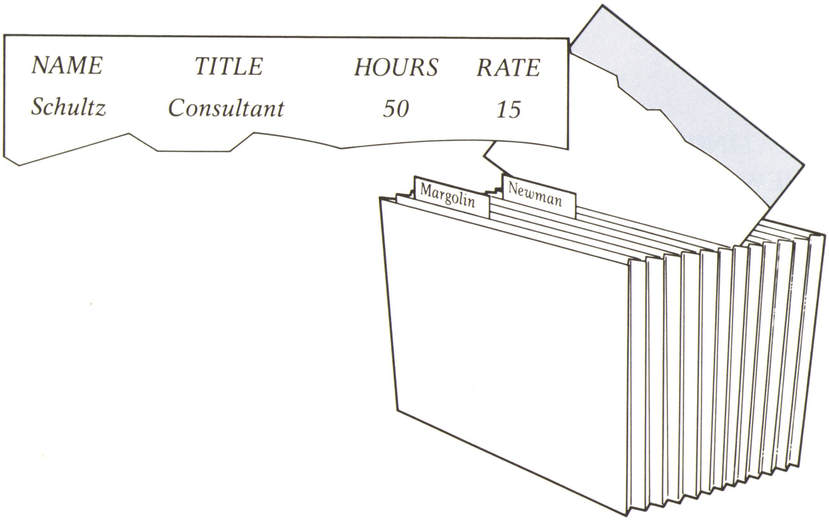

NAME
TITLE
HOURS
RATE
Schultz
Consultant
50
15
Margolin
Newman

FIGURE 16-4

A random-access file is like an accordion file

Since the disk is a mechanical device, it takes time to read information from it. Therefore, programs that have to read lots of extraneous information in order to find some particular record take longer to execute because the sequential file requires much time-consuming disk reading. Ideally, we want our programs to go directly to the record we want, without reading all the preceding records. The only way to accomplish this is with a random-access file.

There is more *programming* involved in using random-access files, but the benefits can be worth it. Suppose the Brothers Four Corporation expanded and hired 500 new employees. We wouldn't want our program to look through a long sequential file to find a particular employee, because that would take lots of time. On the other hand, many programming tasks are more appropriately accomplished with sequential files. For example, text files such as word-processing documents, or a file containing a sequence of musical notes, are both better processed by a program in a beginning-to-end order than in a random order.

Because the subject of files is quite large and complex, we're going to illustrate each step in the various processes concerning the use of files one at a time. Sequential files are the easiest to learn about, so that's where we'll start. We'll lay the groundwork for many of the example programs in this chapter by creating a sequential file that will eventually contain the payroll information for the employees of the Brothers Four Corporation. Later we'll explore the use of random-access files.

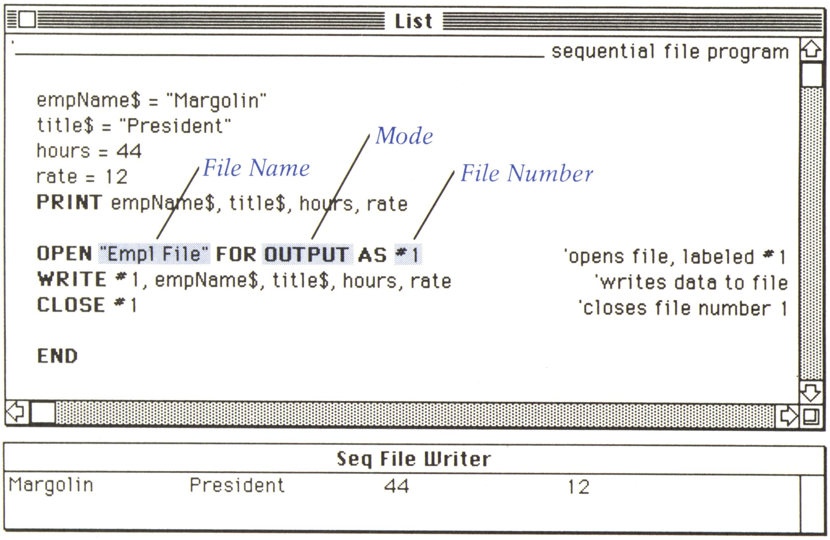
CREATING A NEW SEQUENTIAL FILE

We'll begin with a simple example program that creates the employee file for the Brothers Four Corporation and writes one record in it. There are three steps that are necessary to set up our file:

1. Open a new file called *Empl File* on the disk.
2. Write the employee data to the file.
3. Close the file.

Each of these steps is carried out by a BASIC statement. We'll be referring to the program in Figure 16-5 during the following discussion.

In our sequential-file program, the first four assignment statements simply set up the four different variables that will contain the employee information—name, title, hours worked, and hourly rate. (In a typical program, these values would probably be entered from the keyboard via INPUT statements or edit fields.) The variables are displayed on



```
empName$ = "Margolin"
title$ = "President"
hours = 44
rate = 12
PRINT empName$, title$, hours, rate

OPEN "Empl File" FOR OUTPUT AS #1              'opens file, labeled #1
WRITE #1, empName$, title$, hours, rate          'writes data to file
CLOSE #1                                       'closes file number 1

END
```

```
Margolin         President         44              12
```

FIGURE 16-5

A program to create a new sequential file

the screen by the PRINT statement. The three statements that follow take care of the three file requirements we just mentioned. Let's see how each statement works.

File definition: The OPEN statement

Before our BASIC program can use a file, we must first *define* the file using the OPEN statement. This statement provides BASIC with necessary information about the file. Here's the syntax for the simplest form of the OPEN statement:

OPEN *File Name* **FOR** *Mode* **AS #** *File Number*

Let's see what each parameter in the OPEN statement means.

The File Name parameter

The *File Name* parameter provides BASIC with the name of the file we wish to use in our program. This is specified as a string expression. Our example simply uses a string constant *Empl File* to name the file, but as we'll see, string variables can also be used.

The Mode parameter

Whenever we open a file to be used by our program, we must tell BASIC what kind of operations we're going to be doing with the file. This is the purpose of the *Mode* parameter, which for sequential files can be one of three possibilities:

- OUTPUT
- INPUT
- APPEND

OUTPUT mode means that any data we write into the file in this program *replaces* whatever data might already be in the file. If no file with the same name exists, then the OUTPUT mode will cause BASIC to create a new file on the disk. OUTPUT mode also means that only output operations can be performed on the file while it is open. So this is the mode we want in our example, since all we want to do is *write* one employee's data record to the file.

INPUT mode does just what its name describes: It opens the file so that data can be *input* from the file into the program. If we specify this mode to BASIC, it assumes that the file already exists on the disk. If it doesn't, an error message results.

APPEND mode is like OUTPUT mode in that it is used only for output operations to the file. The difference is that in APPEND mode, BASIC checks to see if the file already exists on the disk. If it does exist, new data is *appended* to the end of the file, leaving the existing data undisturbed. If the file does not exist, then BASIC creates it for us and operates as though we had specified OUTPUT mode.

If no mode is specified, BASIC assumes that we want to open a *random-access file,* and proceeds entirely differently. We will learn all about random-access files later in this chapter. For now, let's finish discussing the OPEN statement.

The File Number parameter

The *File Number* parameter is used to assign a number to the file that BASIC can refer to in subsequent operations in the program. This simply means that we don't have to keep using the *File Name* parameter in other BASIC statements that refer to this file; we can use the number instead. In our example, we've assigned the number 1 to the new employee file we're creating.

Writing data to the file: The WRITE# statement

Now that we have opened a sequential file for our program to use, we can "write" the employee data to the file. To "write" means that the information you want in the file is passed to it or "written" to the disk. This is accomplished by the WRITE# statement, which has the following syntax:

WRITE # *File Number, Expression List*

Let's see what each parameter means.

The File Number parameter

Remember, we said that once a file has been opened in a BASIC program, we use the file number to refer to it. The *File Number* parameter specifies *which file* data will be written to. This doesn't seem important in our program since we're only using one file. But it is possible to use many different files at the same time in the same program, as long as they are each assigned a unique file number when they are opened.

The Expression List parameter

The *Expression List* contains the data that is to be written, as a record, to the file. This can be in the form of string and numeric constants, variables, or expressions. Each item in the expression list constitutes a *data field.* As you can see in our example program, we've used the variable names representing the data that we want written to the file. Notice that the names in the expression list are separated by commas, which is required. In essence, the items separated by commas represent the fields of the single record that will be written.

This single WRITE# statement writes the payroll data for this one employee to the newly created file called *Empl File.* At this point, we could continue to write payroll data for the rest of the employees, but let's go on to discuss the third consideration: closing and saving the file. Later, we'll see how to add the payroll data for the other employees to the file.

Saving the file: The CLOSE statement

When our data has been written to a file, and we no longer intend to process it in the current program, we use the CLOSE statement to close the file. While the function of the CLOSE statement is not noticeable when it is executed, it tells BASIC that our program has finished using the file, that any memory space allocated to handle the file is available for other purposes, and that the file's *File Number* parameter can be reassigned. The syntax of the CLOSE statement is simply:

CLOSE # *File Number*

The *File Number* parameter has the same meaning for the CLOSE statement that it had for the OPEN and WRITE statements: It specifies which file is to be closed.

If you experiment with this program, you will find that it performs quite well without the CLOSE statement. That's because when the program finishes executing, BASIC automatically closes any open files. It's comforting to know that we can't hurt anything if we accidentally forget to include the CLOSE statement in our programs. But we recommend that you always use a CLOSE statement when you are finished working with a file. In larger programs, it provides much better program organization and memory utilization.

BASIC files on the desktop

Let's quit BASIC for a moment and return to the Macintosh desktop. As we can see in Figure 16-6, the new file we've just created appears as a blank document icon in the disk window. We see that the name we assigned in the OPEN statement appears beneath the icon for the file.

The file created by our BASIC program

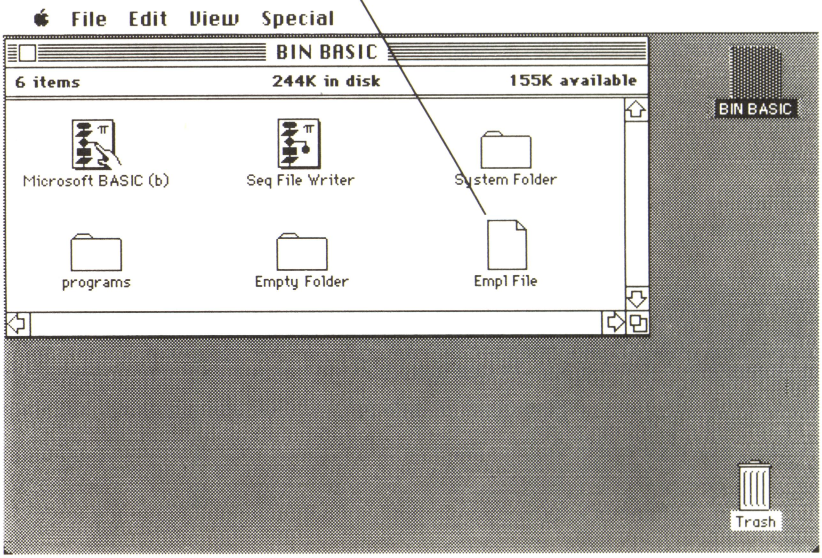

File
Edit
View
Special
BIN BASIC
6 items
244K in disk
155K available
Microsoft BASIC (b)
Seq File Writer
System Folder
programs
Empty Folder
Empl File
BIN BASIC
Trash

FIGURE 16-6

The Macintosh desktop displaying the new file's blank document icons

It's important to understand that the blank document icon is not a program that can be executed by double-clicking its icon. Instead, the icon indicates that it is a *file of data* pertaining to the employees of the Brothers Four Corporation. This file and the data contained in it are now available for use by other programs. Let's write another example program to *read* the data stored in this file.

READING DATA FROM A SEQUENTIAL FILE

It's just as easy to "read" data from a file into our program as it is to write it. To illustrate this, we'll use another example program, shown in Figure 16-7 below, to read the employee's payroll data.

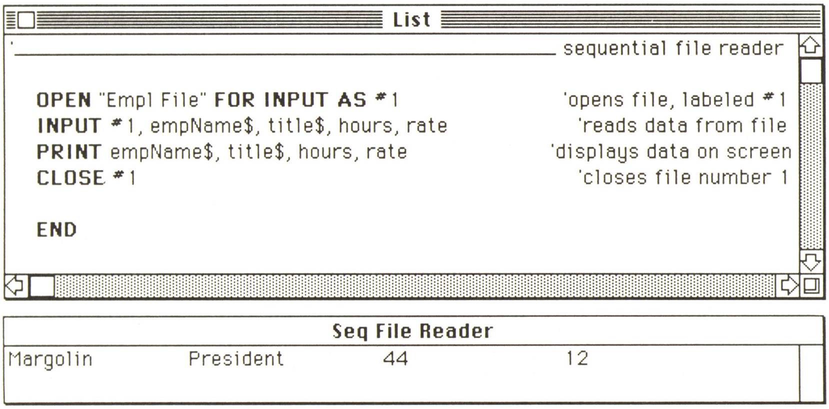
As you can see in the program, it only takes four BASIC statements to read the data we just saved in *Empl File* in our previous program. Let's look at the key factors.

File definition: The OPEN FOR INPUT statement

In our discussion of the OPEN statement in the last program, we learned about the *Mode* parameter. It specifies to BASIC how we intend to use the file in our program. In this case, we want to *read* the employee's data from the file, which is an *input* operation. That's why the OPEN statement in our program specifies INPUT as the mode. Everything else about the OPEN statement is the same as in our previous example.

Reading the data: The INPUT# statement

Since reading data from a file into our program is an input operation, it makes sense that we would use a form of the INPUT statement to do the job. The INPUT# statement works just like the familiar INPUT statement



```
'__________________________________________________ sequential file reader

   OPEN "Empl File" FOR INPUT AS #1                    'opens file, labeled #1
   INPUT #1, empName$, title$, hours, rate              'reads data from file
   PRINT empName$, title$, hours, rate                'displays data on screen
   CLOSE #1                                            'closes file number 1

   END
```

```
Margolin          President          44               12
```

FIGURE 16-7

Reading data from a sequential file

we've used, except that the # sign tells BASIC that the data is to be read from a disk file rather than from the keyboard. Here is the general syntax of the INPUT# statement:

INPUT # *File Number, Expression List*

No doubt you've noticed that the parameters for the INPUT# statement are the same as those for the WRITE# statement in the preceding section. That's because they perform the exact opposite operations on the same file and data.

The *File Number* parameter tells BASIC from which file the data is to be read. In our example, we have only one open file, called *Empl File.* More advanced programs may have more than one file in use at the same time. In that case, each file would be assigned a different file number when it was opened.

The *Expression List* for the INPUT# statement consists of the names of the variables that are to receive the data from the file. In our example, we can see the same list of variable names, in the same order, as in our previous example program which wrote the data to the file. The PRINT statement immediately following the INPUT# statement shows that the values of the variables were set by reading the data from the file.

ADDING DATA TO AN EXISTING SEQUENTIAL FILE

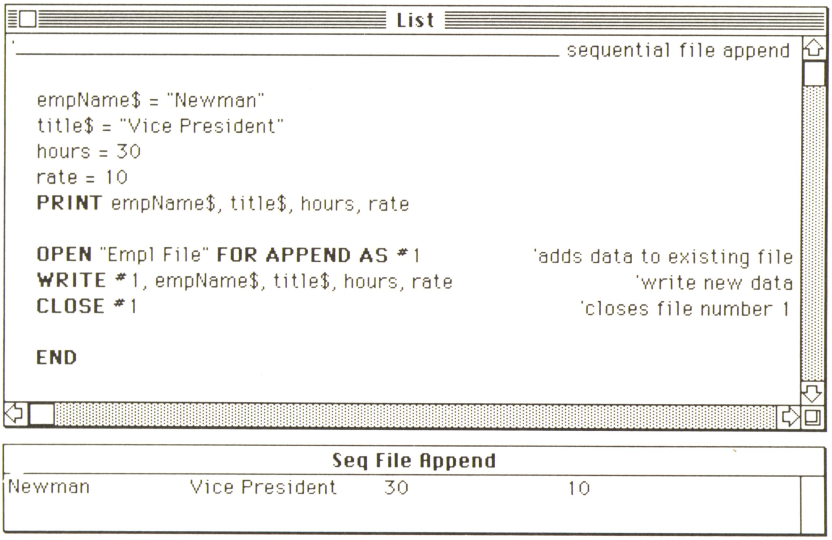
We've used a very simple file, containing only one employee's payroll data, to illustrate the concepts of *creating, writing,* and *reading* a file. But there are four employees of the Brothers Four Corporation, so we'll want to add the other three to our file. This is an opportunity to illustrate the concept of *adding* data to an existing sequential file.

File definition: The OPEN FOR APPEND statement

We'll use another program, shown in Figure 16-8, to demonstrate how data can be added to an existing sequential file. Besides the values that are assigned to the variables and a small change in the OPEN statement, this program is almost identical to the first example program that created the file and wrote the first employee's payroll data to it. The important difference is in the OPEN statement: We've specified the *Mode* parameter as APPEND. This means that we want to open the file named *Empl File* and keep the data that is already stored in it. Any new data that we write to this file will be added at the end of the existing file.

FIGURE 16-8

Adding data to an existing sequential file



```
empName$ = "Newman"
title$ = "Vice President"
hours = 30
rate = 10
PRINT empName$, title$, hours, rate

OPEN "Empl File" FOR APPEND AS #1           'adds data to existing file
WRITE #1, empName$, title$, hours, rate                  'write new data
CLOSE #1                                          'closes file number 1

END
```

```
Newman            Vice President      30                10
```

There is an additional feature of the APPEND mode. If the file name that you specify does not exist, BASIC creates it as a new file, and behaves just as though you had specified OUTPUT as the file mode. This means that the same program can create a file on one occasion, and then add data to that same file on another occasion.

FINDING THE END OF A SEQUENTIAL FILE: THE EOF FUNCTION

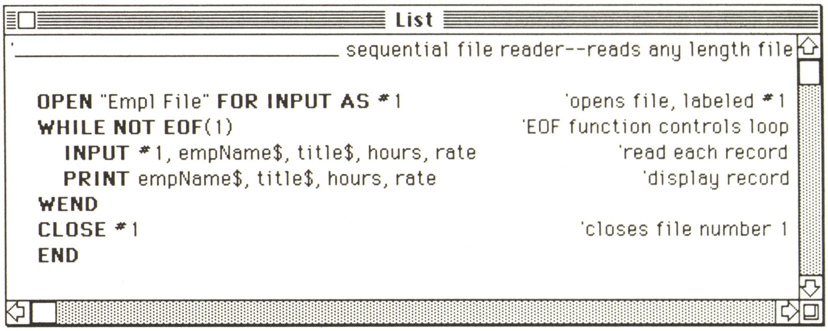
Now that we've added another employee's payroll data to our file, it would be nice to get a listing of the file to make sure everything is stored properly. In order to see the file's contents, we could just write a simple program that contained two INPUT# statements; one to read the first employee's data for display on the screen, and the other to read the second for display. But what will we need to do to list the file when it contains payroll information for 500 employees?

The answer is not 500 INPUT# statements, but rather the EOF function, which is short for *End Of File.* The EOF tells our program when it has reached the end of the file being processed. Figure 16-9 on page 402 shows how we've used the EOF function to find the end of *Empl File.*

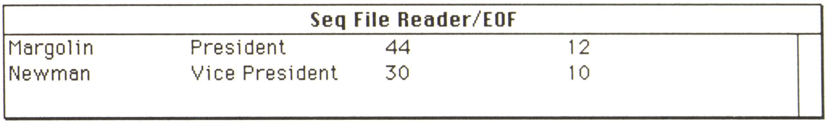
The simple seven-line program in Figure 16-9 displays all the records in *Empl File,* regardless of whether there are two or two thousand. We

FIGURE 16-9

Using the EOF function to detect the end of a file



```
'_____________________________ sequential file reader--reads any length file

 OPEN "Empl File" FOR INPUT AS #1                       'opens file, labeled #1
 WHILE NOT EOF(1)                                     'EOF function controls loop
   INPUT #1, empNname$, title$, hours, rate                      'read each record
   PRINT empName$, title$, hours, rate                            'display record
 WEND
 CLOSE #1                                                  'closes file number 1
 END
```



```
Margolin      President        44            12
Newman        Vice President   30            10
```

start with an OPEN statement, specifying INPUT mode, so we will be reading data from the file.

The WHILE...WEND loop that follows uses the EOF function to control its operation. We'll explain how this works in a moment. Inside the loop there is an INPUT# statement. This reads the data from the file into the variables specified in the list. Remember the "reel of paper tape" analogy? This is like slowly unwinding the tape, reading each data field as we go. There is also a PRINT statement inside the loop to display the variables on the screen, so we can see that it is working.

The EOF function has only one argument: the file number of the open file that we want to test for the end-of-file condition. The syntax is:

EOF(*File Number*)

Like all functions, EOF returns a value. It is this value that signals whether the last record has been read and the end of file has been reached. The two possible values are:

0—False condition (end of file has not been reached)
1—True condition (end of file has been reached)

Each condition of truth or falseness in BASIC has a numeric value associated with it. Logical falseness is tied to the number 0, and truth is specifically assigned the value −1 (although any non-zero number is evaluated as true). If we type in the command window:

```
PRINT 3 > 2
```

BASIC displays the number −1 in the Output window. This occurs because BASIC has evaluated the logical expression 3 > 2 as true, and the PRINT command displays true's numeric value, −1.

Programmers will often use a construct that has a form similar to the following:

```
WHILE -1
  PRINT "doing something"
WEND
```

Since the number −1 has the logical value of truth, this loop will repeat indefinitely. When we are experimenting, this programming structure, called an *endless* or *infinite* loop, can be a convenient way to keep looping when we don't know when the loop is supposed to end, but it is considered inelegant. The only way to end this kind of loop is to interrupt the program—regardless of what it is doing—by some artificial means, like choosing Stop from the Run menu. A better way to end an infinite loop is to establish a variable that is set to true at the beginning of the program and then made false by some condition *inside* the loop. For example:

```
looping = -1
WHILE looping
  INPUT "keep looping?", answer$
  IF answer$ = "yes" THEN looping = 0
WEND
```

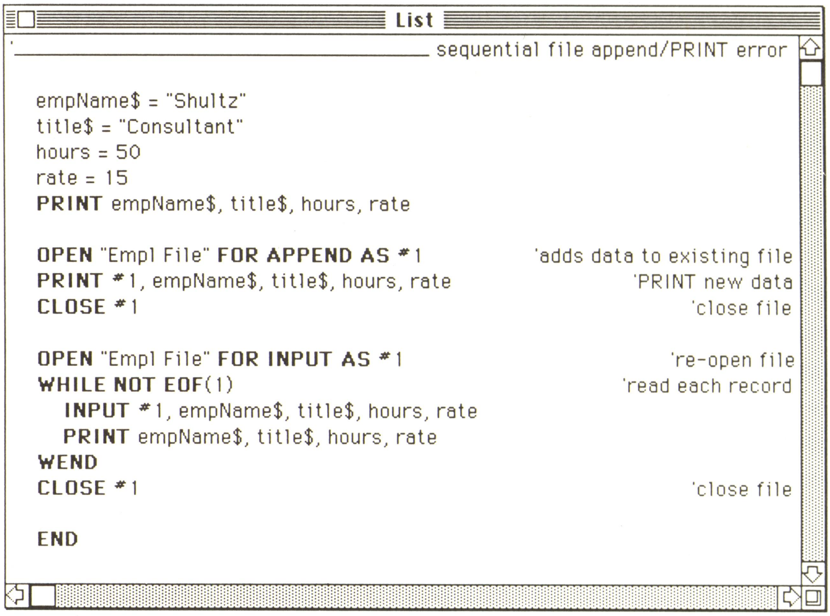
In the case of our new file-reading program, as each record in the file is read, the EOF function is updated with a value that depends on whether there are more records in the file. While there are records, EOF(1) is *false*, and the logical expression NOT EOF(1) is *true*. When the program reads the last record in the file, the EOF function is set to *true*, the expression NOT EOF(1) becomes *false*, and the loop ends.

One side note here—if you ran the program in Figure 16-8 more than once, a record was added to the file each time you ran it. So, if the Output window of the current program shows more than one reference to Newman, this is undoubtedly what happened.

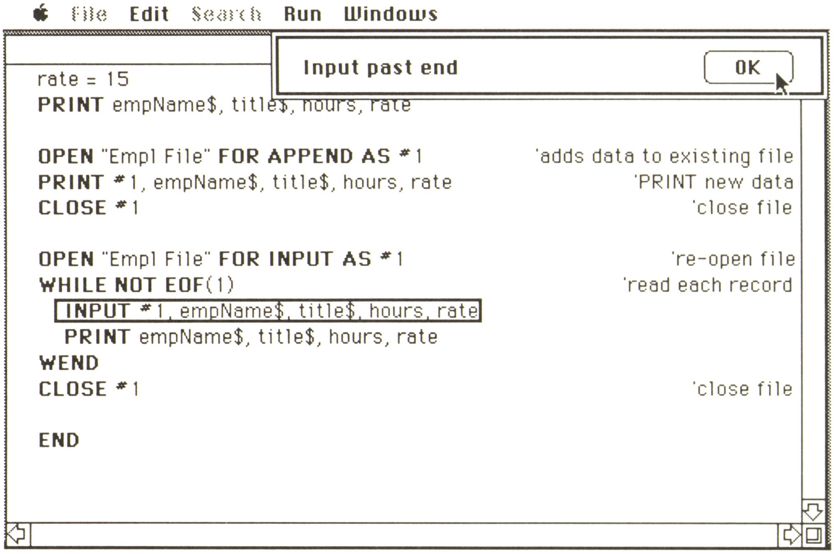
Since the EOF function returns either a true or false value, it's ideal for use in a WHILE...WEND loop, where some true or false condition can control the operation of the program. We'll see this structure many times in programs that use files, because it is so easy to use.

USING PRINT# INSTEAD OF WRITE#

You may have noticed in your Microsoft BASIC manual that there is a PRINT# statement that can be used to write data to a sequential file. While you can use this statement instead of the WRITE# statement, there is a significant difference in its operation. Let's use another example program to demonstrate this difference. We'll append a third employee's data to *Empl File,* but change WRITE #1 to PRINT #1 to write it to the file. The new listing and sample output is shown in Figure 16-10.



```
'____________________________________ sequential file append/PRINT error

empName$ = "Shultz"
title$ = "Consultant"
hours = 50
rate = 15
PRINT empName$, title$, hours, rate

OPEN "Empl File" FOR APPEND AS #1               'adds data to existing file
PRINT #1, empName$, title$, hours, rate                      'PRINT new data
CLOSE #1                                                          'close file

OPEN "Empl File" FOR INPUT AS #1                               're-open file
WHILE NOT EOF(1)                                          'read each record
  INPUT #1, empName$, title$, hours, rate
  PRINT empName$, title$, hours, rate
WEND
CLOSE #1                                                          'close file

END
```

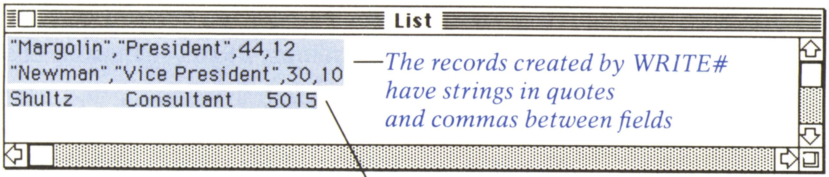


Input past end

```
rate = 15
PRINT empName$, title$, hours, rate

OPEN "Empl File" FOR APPEND AS #1               'adds data to existing file
PRINT #1, empName$, title$, hours, rate                      'PRINT new data
CLOSE #1                                                          'close file

OPEN "Empl File" FOR INPUT AS #1                               're-open file
WHILE NOT EOF(1)                                          'read each record
  INPUT #1, empName$, title$, hours, rate
  PRINT empName$, title$, hours, rate
WEND
CLOSE #1                                                          'close file

END
```

FIGURE 16-10

The "Input past end" error message that results when we use PRINT# instead of WRITE#



```
"Margolin","President",44,12
"Newman","Vice President",30,10
Shultz      Consultant    5015
```

—*The records created by WRITE# have strings in quotes and commas between fields*

The record created by PRINT# looks just like it would be displayed in the Output window

FIGURE 16-11

Two records created with WRITE#, which uses delimiters, and one created with PRINT#, which doesn't

As we can see, for some reason BASIC doesn't like our program any more. It worked fine, though, before we added the third record using the PRINT# statement, so that must be causing our program problems.

If you ran the program you noticed that only the first two records were displayed, which adds weight to our suspicions about the PRINT# statement. The error message says that our program tried to "Input past [the] end" of the file. Let's see if we can understand what happened.

Without going into a big technical discussion here about the way that data is stored in a file, let's understand that all sequential files created by BASIC are *text* files. The data is stored in the form of ASCII characters, just like our BASIC programs themselves when we say "save as" text. Because of this, we can open a text file with the BASIC interpreter just as though it were a program and examine its contents. We can open *Empl File* by choosing Open from the File menu, and then selecting *Empl File* in the Open dialog box. Once the file is opened, its contents appear in BASIC's List window, as shown in Figure 16-11.

Here's where we begin to see the difference between WRITE# and PRINT#. The first two lines in the List window are the two records that were written to the file using the WRITE# statement. The third line is the record that was written to the file using the PRINT# statement. The obvious difference is that PRINT# does not use *delimiters* when it writes the data to the file.

Delimiters are characters that are automatically included by the WRITE# statement to separate data fields. String fields containing the employee name and title are automatically enclosed in quotes, and there are commas separating all four of the data fields. Their purpose is to specifically define the difference between one data field and the next so that when we use INPUT$ it will know what to do. The record that was written using the PRINT# statement appears almost as though it had been output to the screen by an ordinary PRINT statement. There are no delimiters between the fields.

Since BASIC puts a handy box around statements that cause error messages, we can see that the error occurred with the statement:

```
INPUT #1, empName$, title$, hours, rate
```

Because there were no delimiters in the data, the entire third record was seen by BASIC as one string, and input into the variable *empName$*. So, when BASIC tried to input a value for the next variable *title$*, there was nothing remaining to read but the end of the file, and the error occurred.

When PRINT# is useful

The PRINT# statement is useful in some applications that involve only lines of text (like word processors) because it does not include delimiters the same way that WRITE# does. You are responsible for your own delimiters when you use PRINT#. For our purposes, however, we'll use only WRITE# to write data to our files. In this way, we'll know that all the data fields are properly separated.

We'll be returning to the *Empl File* later, when we'll write a complete program that really creates it correctly. Right now, let's discover a convenient way of specifying file names in our programs.

FILE NAMING AND SELECTION: THE FILES$ FUNCTION

The example programs we've been using in this chapter have all used the same file. Therefore, it was simple to specify the *File Name* parameter in the OPEN statement with just the string constant *Empl File*. In the real world, you may want your programs to be much more flexible.

For example, let's suppose that we are developing a payroll program that will be used to process the payroll for several different companies. We will want to create a separate file for each company's payroll information, and be able to choose which file our program will use. All of this is possible using the FILES$ function, which performs these two powerful operations:

- It accepts user-typed entry of a file name.
- It allows the user to select a file name from a scrolling window.

Entering a file name: The FILES$(0) function

There are two forms of the FILES$ function. The first form of FILES$ has the following syntax:

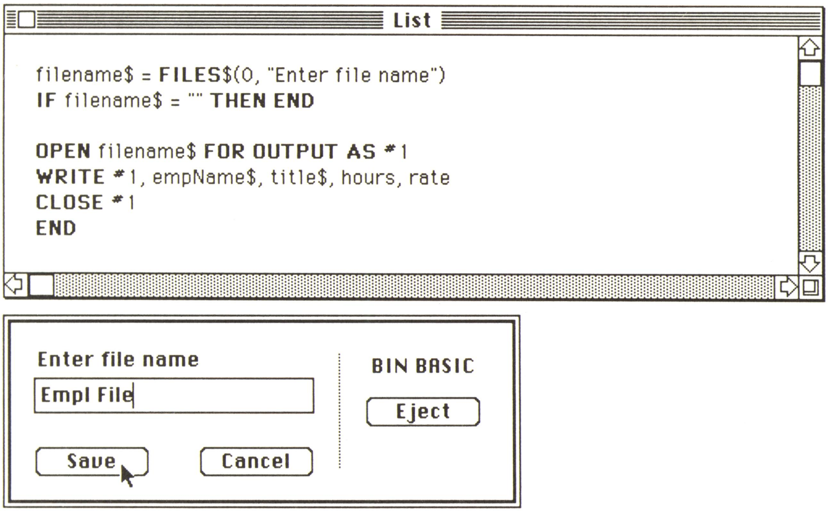
File Name = **FILES$(0**, *Prompt String*)

Used in this form, the FILES$ function automatically produces the dialog box shown in Figure 16-12.

As we can see in the figure, this is a powerful function! With one statement we can create the familiar Save dialog box that we've seen so many times before in other Macintosh programs. The dialog box comes complete with all the buttons to eject the disk, change the drive (if an external drive is hooked to the Mac), or cancel the whole operation—all handled automatically by BASIC. The Save button is a little misleading, though, because the FILES$ function doesn't actually save anything in a file; it only allows the user the incredible Macintosh flexibility we've come to expect in the way a file name is entered.

The *Prompt String* parameter of the FILES$(0) function contains an optional prompt that is displayed in the dialog box directly above the edit field where the file name is to be entered. Notice in our example program, the prompt string produces the prompt *Enter file name.*

Also notice the IF...THEN statement in Figure 16-12. If the user clicks the Cancel button in this dialog box, the *filename$* value returned by FILES$ is the null string (""). Although we're not including the BASIC statements in our example here, a real program could have a section la-



```
filename$ = FILES$(0, "Enter file name")
IF filename$ = "" THEN END

OPEN filename$ FOR OUTPUT AS #1
WRITE #1, empName$, title$, hours, rate
CLOSE #1
END
```

Enter file name

Empl File

FIGURE 16-12

The dialog box created by the FILES$(0) function to allow the user to enter a file name beled Cancel to handle this eventuality. If this sort of file-name control is not handled by a program, BASIC tries to open a file without a name, resulting in an error message.

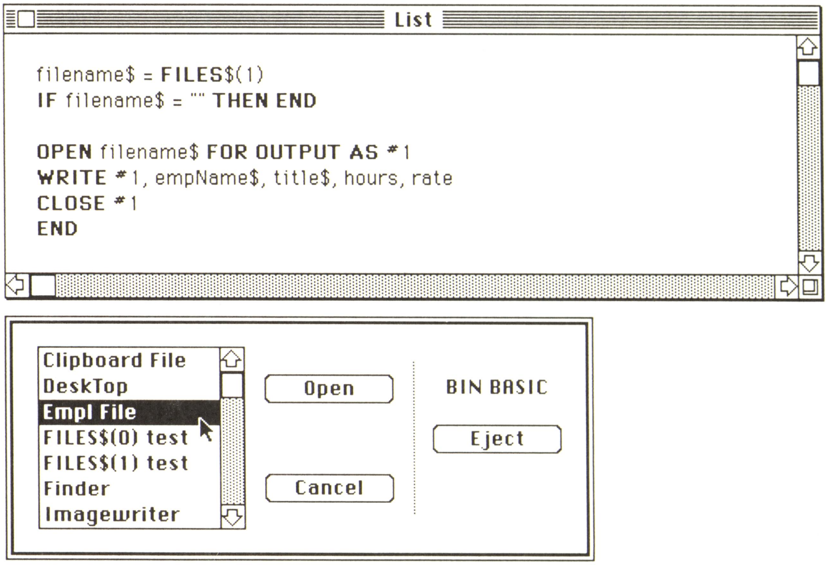
Selecting a file name: The FILES$(1) function

The second form of the FILES$ function allows the user to select the name of an *existing* file from the familiar scrolling dialog box shown below in Figure 16-13.

Here is another powerful tool in our Microsoft BASIC workshop! With one statement containing the FILES$ function, we've created a dialog box, complete with buttons and a scrolling window. All the user has to do is scroll to the desired file name, and click on it with the mouse. This causes the name to be highlighted, as shown in the figure.

When the user clicks the Open button with the mouse or simply double-clicks on the file name, the file name that is selected is returned and put into the string-variable *filename$* in our program. Here again, if the user clicks the Cancel button the FILES$ function returns a null string, which is used in the IF...THEN statement to cause the program to end.

You'll notice that there are a lot of different file names displayed in our example, including *Clipboard File, DeskTop,* and others that are not relevant to our particular payroll application. To simplify the selection that must be made by the user, we can assign a four-character *File Type* to our files. Then, we can specify that only files of a certain type appear in the scrolling window. Let's see how we do this.



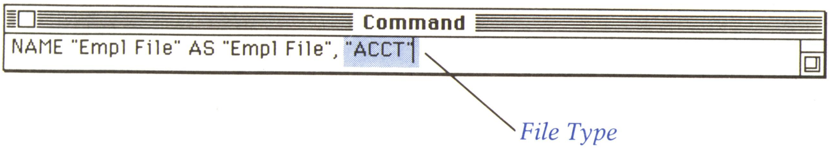
```
filename$ = FILES$(1)
IF filename$ = "" THEN END

OPEN filename$ FOR OUTPUT AS #1
WRITE #1, empName$, title$, hours, rate
CLOSE #1
END
```

FIGURE 16-13

The dialog box created by the FILES$(1) function to allow the user to select a file name

FIGURE 16-14

Setting a File Type using the Command window



```
NAME "Empl File" AS "Empl File", "ACCT"
```

File Type

Changing the file name and type: The NAME statement

The NAME statement allows us to both change the name of a file, and also assign a *File Type* to a file, which allows us to display only the names of files of a certain type in the file-name window. Normally, setting the *File Type* would be done in the program that initially creates the file. But this gives us a perfect opportunity to use the Command window to execute this one statement immediately, as shown in Figure 16-14 above.

The general syntax of the NAME statement is:

NAME *Old Name* **AS** *New Name, File Type*

Notice that, besides being able to assign a *File Type* parameter to a file, we can actually change the entire name of a file with the NAME statement. The way we've used it in Figure 16-14, the file name remains the same, but the *File Type* is added.

The *File Type* must be a string of not more than four characters. In our example, we've assigned ACCT to the *File Type* parameter, since the employee file will be part of an "accounting system."

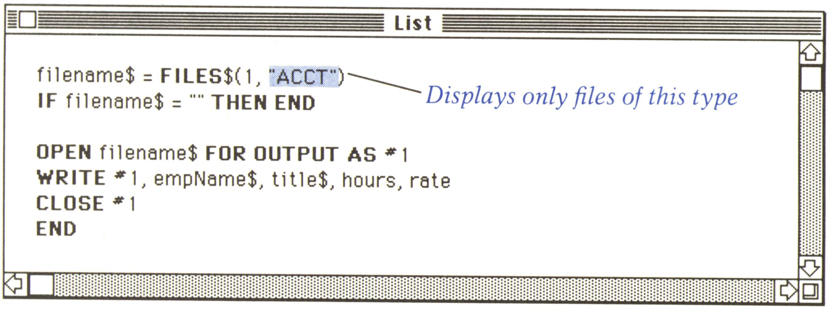
Now, let's integrate what we've just learned about the *File Type* with what we learned earlier about the FILES$ function.

Specifying the File Type with FILES$(1)

The example BASIC code that is shown in Figure 16-15 illustrates how we can add a *File Type* parameter to the FILES$(1) function. This causes the FILES$ function to display only the names of files that have a matching

FIGURE 16-15

Using the FILES$(1) function to specify a File Type



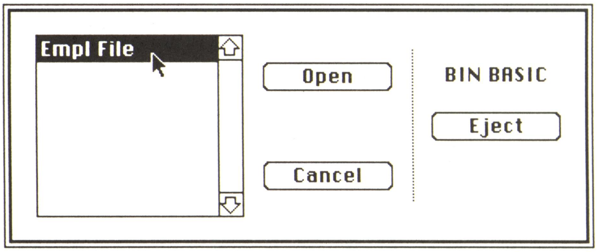
```
filename$ = FILES$(1, "ACCT")
IF filename$ = "" THEN END

OPEN filename$ FOR OUTPUT AS #1
WRITE #1, empName$, title$, hours, rate
CLOSE #1
END
```

Displays only files of this type

(continued)

FIGURE 16-15

(continued)



File Type parameter. As we desired, only the name *Empl File* is displayed in the scrolling window.

ALL-IN-ONE PROGRAM

Remembering that we've screwed up our employee file by experimenting with the PRINT# statement, we'll have to start over. So let's take a short break to quit to the Macintosh desktop and trash the *Empl File* we've been using. When we restart BASIC we'll have a good opportunity to use many of the concepts and BASIC tools we've learned. We'll write a new example program that:

- Presents the file-name scroll box.
- Accepts an alternate file name from the keyboard in case the file we want to use doesn't exist yet.
- Creates the *Empl File* and assigns file type.
- Accepts employee data from the keyboard and writes it to the file.
- Reopens the file for reading employee data.
- Calculates and displays earnings for each employee.

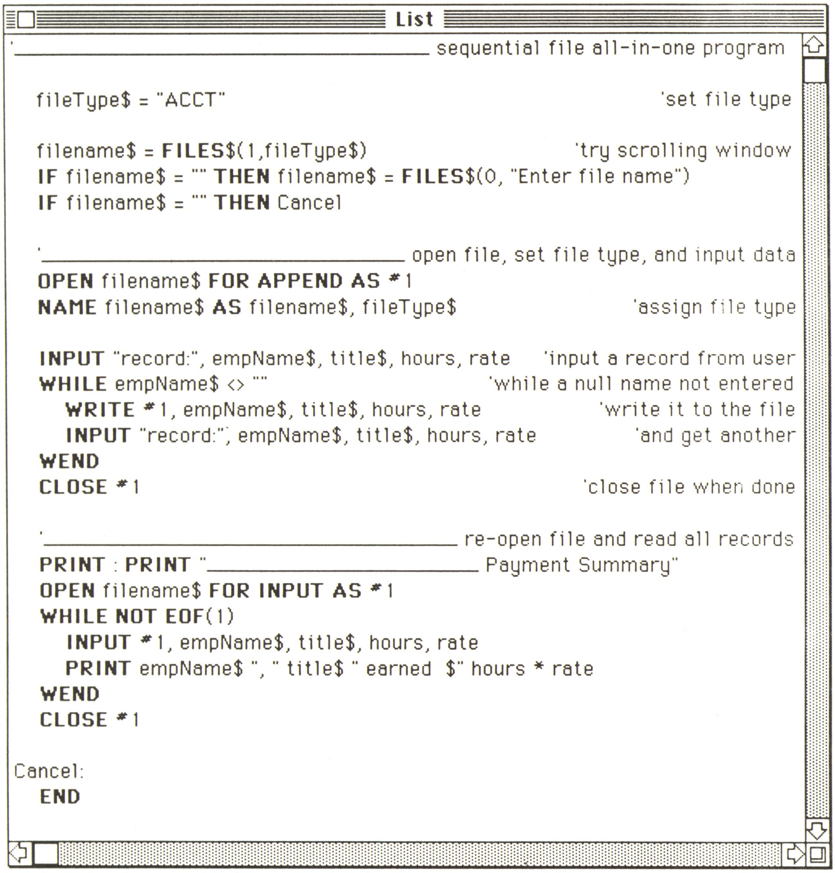
In Figure 16-16, you'll see our miracle program and an example of its output. Let's go through the program section-by-section.

In the first part of the program we use a combination of the FILES$(0) and FILES$(1) functions so the user can either select a file name from the scrolling window, or type one in the edit field. Notice that the file is opened for APPEND. This means that any data that was already in the file is retained, and the new data will be added at the end of the file. Since we've trashed the *Empl File* in order to start over, the first dialog box created by the program is empty. Just click on Cancel to go to the second box and enter the name there.

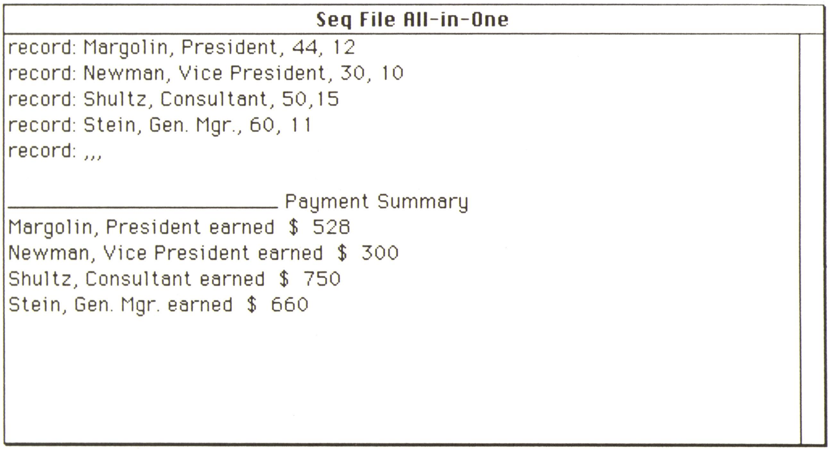
The next section of the program is pretty straightforward. We've included a method for telling the program when we're done entering data. Entering three commas with nothing else causes all four variables in the

FIGURE 16-16

An earnings-summary program and sample output



```
'_______________________________ sequential file all-in-one program

  fileType$ = "ACCT"                                    'set file type

  filename$ = FILES$(1,fileType$)                  'try scrolling window
  IF filename$ = "" THEN filename$ = FILES$(0, "Enter file name")
  IF filename$ = "" THEN Cancel

  '____________________________ open file, set file type, and input data
  OPEN filename$ FOR APPEND AS #1
  NAME filename$ AS filename$, fileType$                'assign file type

  INPUT "record:", empName$, title$, hours, rate    'input a record from user
  WHILE empName$ <> ""                           'while a null name not entered
     WRITE #1, empName$, title$, hours, rate            'write it to the file
     INPUT "record:", empName$, title$, hours, rate         'and get another
  WEND
  CLOSE #1                                            'close file when done

  '________________________________ re-open file and read all records
  PRINT : PRINT "_______________________ Payment Summary"
  OPEN filename$ FOR INPUT AS #1
  WHILE NOT EOF(1)
     INPUT #1, empName$, title$, hours, rate
     PRINT empName$ ", " title$ " earned  $" hours * rate
  WEND
  CLOSE #1

Cancel:
  END
```



```
record: Margolin, President, 44, 12
record: Newman, Vice President, 30, 10
record: Shultz, Consultant, 50,15
record: Stein, Gen. Mgr., 60, 11
record: ,,,

_______________________ Payment Summary
Margolin, President earned  $  528
Newman, Vice President earned  $  300
Shultz, Consultant earned  $  750
Stein, Gen. Mgr. earned  $  660
```

expression list of the INPUT statement to have a null value (either 0 or ""). When the user types in three commas, the program falls through the WHILE...WEND loop and the file is closed.

In the next section the file is reopened, this time for INPUT. To continue our "paper-tape" analogy, the effect of closing and reopening a file is like rewinding the tape back on its roll. The WHILE...WEND loop here is similar to the one we've already used to display all the records in the file. Remember that the EOF function is *false* until our program has read all the data in the file, at which time it becomes *true*. By inverting this logic with the NOT function, we can keep the program in the loop until all the data has been read from the file and displayed.

Finally, we find the dauntless CLOSE statement, which we always include for good measure to make sure a file is properly closed.

The *Cancel* label simply marks the END of the program. This is where the program branches if the user doesn't enter a file name or clicks Cancel when prompted for a file name at the beginning of the program.

Sequential files summary

This concludes our study of sequential files. While we've only used one file and some very simple example programs, we've seen that powerful things can be done with files using Microsoft BASIC. With a minimum of BASIC statements, we can create a file, write data to it, and save it on the disk.

This ability really broadens the horizons of our BASIC programs. It means that we can process huge amounts of data with a small program because the data is stored on the disk rather than in DATA statements in the program itself. For the same reason, we can also pass data back and forth between many different programs. In fact, it's quite possible that your particular programming needs will be perfectly satisfied by the easy-to-use sequential files.

However, the restriction to serial access of the data can be a serious impairment, especially if our program is going to use a large data file in which it will have to "look up" certain information. This could be the case in our payroll program, if, let's say, we needed to find out how many hours a certain employee worked. Using a sequential file, we'd have to start at the beginning of the file and keep reading data until we found the employee we were looking for. This could take a lot of time with a large file, resulting in the overall degradation of our program's execution speed. That's why BASIC supports random-access files, which are explained in the next section.

RANDOM-ACCESS FILES

As its name suggests, a random-access file is one in which any record may be directly used by a program, without having to go through all the preceding records in the file. Earlier in this chapter we drew an analogy between random-access files and "accordion" files that store manila file folders in a particular arrangement. Another analogy, a little closer to home (in the kitchen, actually), is an index-card recipe file. In it, the recipes are organized alphabetically. When we get a sudden craving for beef stroganoff, for example, we go right to the *S* section, pull out the card, and start processing dinner.

Each index card in the recipe file is like a *record* in a disk file. And just as we have to know what we want for dinner before we can look up the proper card, we must also tell BASIC which record we want. But instead of being organized alphabetically, records in random-access files are organized *numerically,* so we need to tell BASIC the number of the record we want. Let's see how records are numbered.

Record numbers and the random-file buffer

We've been talking so much about "records" because the data contained in a random-access file must be arranged in records of equal size. Having all records of equal size is the key to how BASIC can access a particular record for us. For example, if our program tries to access the third record in a random-access file, BASIC calculates exactly how much of the file is occupied by the first two records and skips over that space, going directly to the beginning of the third record.

In order for BASIC to do this, we must establish the *record length* for each random-access file we use. Furthermore, we must specify the arrangement of the data fields within the record. That's because BASIC uses a feature known as the *random-file buffer* whenever programs use random-access files. The random-file buffer is like a "window" through which our program can access any record in a file. We must tell BASIC how large to make this window, and what to expect to "see" through it.

Figure 16-17 on the next page depicts a random-access version of our employee file. We're again representing the file as a *list of data,* as we did at the beginning of this chapter. The records in the file are numbered in ascending order, and each record is exactly the same size. Also shown is a representation of the random-file buffer which we've established as being the length of one record.

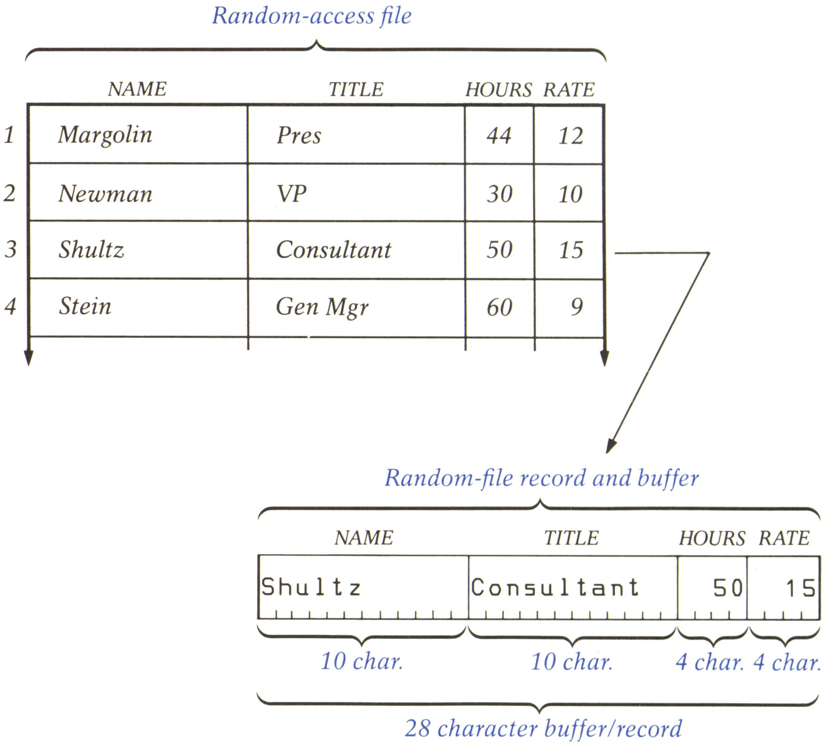

Random-access file
NAME
TITLE
HOURS
RATE
1
Margolin
Pres
44
12
2
Newman
VP
30
10
3
Shultz
Consultant
50
15
4
Stein
Gen Mgr
60
9
Random-file record and buffer
NAME
TITLE
HOURS
RATE
Shultz
Consultant
50
15
10 char.
10 char.
4 char.
4 char.
28 character buffer/record

FIGURE 16-17

The random-file buffer can hold any specific record

As the figure shows, we've specified the length of each of the data fields in the record. Notice that we've set both the *name* and *title* fields to a length of 10 characters. That should be enough for anything we need to enter there. We've also specified two 4-character fields to contain the numeric values for *hours* and *rate.* (This may seem strange now, but later we'll explain more about why we use four characters for these numeric fields.) If we add up the total number of characters required for all the fields in our record, we have a record length of 28 characters.

Now that we have a general feeling of what random-access files are all about, let's see how these concepts are applied to BASIC.

OPENING A RANDOM-ACCESS FILE

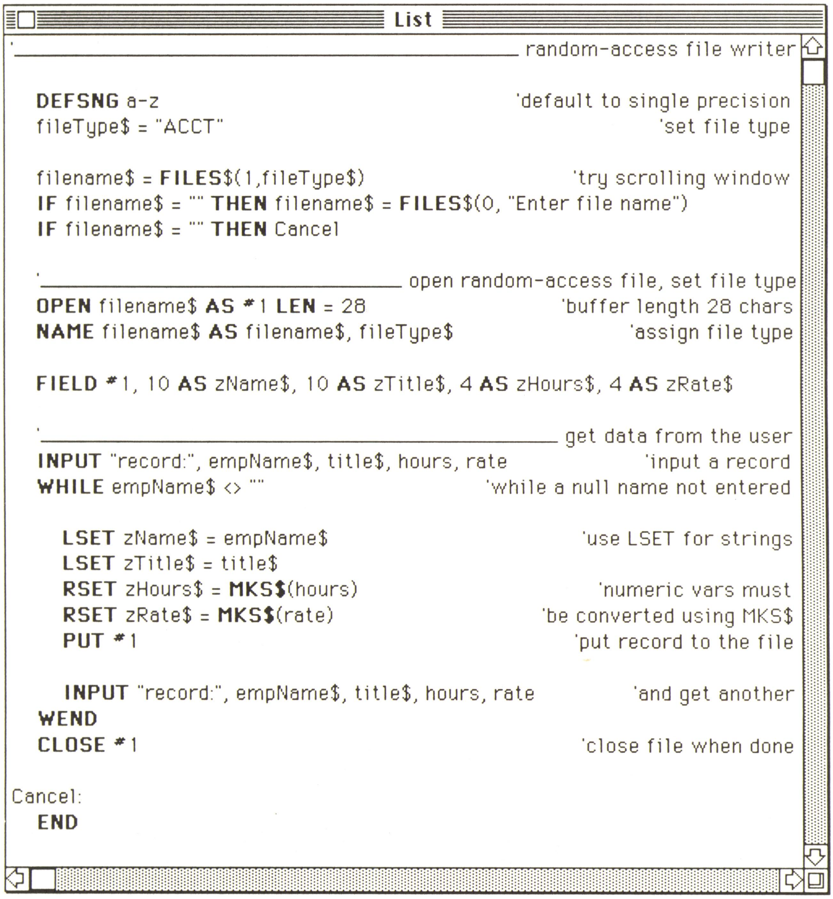
Using random-access files in our programs is more complicated than using sequential files. There are several new BASIC statements and functions that we are going to learn about in the remainder of this chapter. To make things as simple as possible, we're going to use the same employee file concept that we're already familiar with, but this time we will create it as a random-access file.

File definition: The OPEN statement

The first thing we need to do is write a program to create a new random-access file. Then we'll enter all the payroll data. Refer to Figure 16-18 throughout the following discussion.

Our program first assigns the default variable type to "single precision" using the DEFSNG statement. This is important, because it determines how many characters of the random-file buffer will be required to contain the numeric variables. We'll see how this works in a moment.

We've used the FILES$ functions to allow the user to either select the file name or enter it from the keyboard. Always remember to provide some means of "quitting" if the user clicks the Cancel button in the Open dialog box. This section is identical to the section of the last sequential-file program in Figure 16-16.



```
'_______________________________________________ random-access file writer

  DEFSNG a-z                                       'default to single precision
  fileType$ = "ACCT"                                              'set file type

  filename$ = FILES$(1,fileType$)                         'try scrolling window
  IF filename$ = "" THEN filename$ = FILES$(0, "Enter file name")
  IF filename$ = "" THEN Cancel

  '_______________________________ open random-access file, set file type
  OPEN filename$ AS #1 LEN = 28                          'buffer length 28 chars
  NAME filename$ AS filename$, fileType$                       'assign file type

  FIELD #1, 10 AS zName$, 10 AS zTitle$, 4 AS zHours$, 4 AS zRate$

  '_____________________________________________ get data from the user
  INPUT "record:", empName$, title$, hours, rate                 'input a record
  WHILE empName$ <> ""                              'while a null name not entered

    LSET zName$ = empName$                                  'use LSET for strings
    LSET zTitle$ = title$
    RSET zHours$ = MKS$(hours)                                  'numeric vars must
    RSET zRate$ = MKS$(rate)                              'be converted using MKS$
    PUT #1                                                'put record to the file

    INPUT "record:", empName$, title$, hours, rate               'and get another
  WEND
  CLOSE #1                                                 'close file when done

Cancel:
  END
```

FIGURE 16-18

A program that creates a random-access file

In looking at the OPEN statement in our example, we see something missing, and something new. Here is the general syntax of the OPEN statement when used with random-access files:

OPEN *File Name* **AS** # *File Number* **LEN** = *File-Buffer Size*

There is no *Mode* parameter (INPUT, OUTPUT, or APPEND) specified in the OPEN statement, as we saw with sequential files. That's how BASIC knows that we want this new file to be a random-access file. A random-access file doesn't need a *Mode* parameter because it isn't limited to either input or output. Since we can access *any* record in the file without touching the others, we can either write data into that record, or read data from it.

The new addition to the OPEN statement, the keyword LEN, is where we specify the *File-Buffer Size*. Since all input and output operations on this file will be passed through the random-file buffer, this length assignment should also be the size of our record. This is different from the LEN function, which is used to find the length of a string. In our program, we've assigned 28 to LEN, since we have a record length of 28 characters.

Field definition and buffer variables: The FIELD statement

The next step in setting up a random-access file is to define each of the fields in the record (we didn't have to do this with sequential files). This is done using the FIELD statement, as shown in Figure 16-18. The general syntax of the FIELD statement is:

FIELD # *File Number, Field Width* **AS** *String Variable Name,...*

The ellipses indicate that more than one *AS String Variable* can be included in the FIELD statement.

We must have one FIELD statement for each file that we are going to use in our program. Our program only uses one file, so therefore only one FIELD statement is needed. The *File Number* parameter tells BASIC for which file we are going to be specifying the fields.

The FIELD statement must define the size of each of the fields in the record, and assign a special name by which these fields must be referenced. The *Field Width* parameter specifies the size of a field, and the *String Variable* parameter assigns the name.

Buffer variables are not string variables

We've assigned unusual variable names to the buffer variables defined in the FIELD statement. We did so because we want to make sure that we don't inadvertently use a variable name defined in the FIELD statement in some other part of the program where it might be redefined. Here's why.

The variables that are defined in a FIELD statement are named as string variables because they are represented in the buffer as strings of characters. But they are not exactly the same as real string variables, and we should take a moment to discuss the differences.

A buffer variable, or *fielded variable* as they are sometimes called, is a special name that we assign to fields in order to access the random-file buffer and obtain a certain number of characters. A buffer variable should only be used with the statements that we are going to be learning in this section. If you use a buffer variable in an INPUT statement, or an assignment statement, BASIC will redefine the variable as a regular string variable, and we'll lose access to that field in the random-file buffer.

Numeric variables must be handled as strings

The FIELD statement in our example program also specifies the two fields (*zHours$* and *zRate$*) that will be used by the two numeric variables in our Employee record (*hours* and *rate*).

Notice that the buffer variables for these numeric values still have a name ending with a dollar sign. That's because numeric values must first be converted to a string before they are written to a random-access file. As we'll see in a moment, there are BASIC functions that take care of this conversion for us. The important thing to understand is that the string form used to save numeric data in a random-access file is not exactly the same as a string variable. Here's how it works:

- Integer variables convert to two-character strings
- Single-precision variables convert to four-character strings
- Double-precision variables convert to eight-character strings

Remember, we used DEFSNG at the beginning of the program to set the default to single precision, which is why we assigned a field width of 4 to each of these buffer variables.

As we've discovered, BASIC lets us use several types of variables, including string, integer, logical, and the normal numeric or *floating point* numbers. Within this last category there's a further subdivision, between *single-* and *double-precision* floating-point numbers. In binary BASIC, arithmetic using single-precision numbers yields answers accurate to seven significant digits, and each number requires 4 bytes of storage in the Mac's memory. That is why the field width is 4. Double-precision numbers, on the other hand, are accurate to 16 significant digits, but calculations using them take much longer to perform, and require 8 bytes of memory to be stored. A complete discussion of variable types is found in the Microsoft BASIC manual.

WRITING TO A RANDOM-ACCESS FILE

We've opened a new file for random access, and defined all the buffer variables that will provide access to the fields in the random-file buffer. In this section, we'll see how to write a record of data into this file. Continue to refer to Figure 16-18 during the following discussion.

Going from the program to the buffer: The LSET and RSET statements

There are two steps in writing data to a random-access file:

1. Set up the buffer variables to contain the data
2. Write the buffer data to the disk file

Remember that earlier we said that buffer variables are not the same as regular string variables. For that reason, we can't just use an assignment statement to give them a value. Instead, we must use one of the two special BASIC statements, LSET or RSET, to give values to buffer variables. Both statements move data into the random-file buffer so it can be written to the file. The only difference is that LSET left justifies the data in the buffer-variable string, while RSET right justifies the data. Here is the general syntax for both statements:

LSET *Buffer Variable = String Expression*
RSET *Buffer Variable = String Expression*

Moving string data to the buffer

Notice in Figure 16-18 that to move the employee's name and title (two string variables) into the random-file buffer, we must use the two LSET statements, we cannot use a simple assignment statement. Also remember that the buffer variables *zName$* and *zTitle$* were defined in the FIELD statement with a fixed length of 10 characters. If the contents of the variables *empName$* or *title$* are larger than 10 characters, the LSET statement will drop the excess characters from the right end of the string. If we used RSET, excess characters would be dropped from the left end of the string.

Converting numeric data to strings: The MKI$, MKS$, and MKD$ functions

Earlier, we said that we couldn't write numeric data to a random-access file until we converted it to a string form. This is accomplished by the following functions:

MKI$(Integer Expression)
MKS$(Single-precision Expression)
MKD$(Double-precision Expression)

Since our program uses only *single-precision* numeric variables, we're using the MKS$ function to convert the numeric variables *hours* and *rate* to the string form they must be in before they can be assigned to a random-file buffer variable. We then use the RSET statement to right justify the string value returned by the MKS$ function in the buffer variables *zHours$* and *zRate$*.

Moving data from the buffer to the file: The PUT# statement

We have now set up the buffer variables to contain the data that was entered from the keyboard. All that remains is to tell BASIC to actually write the record to the disk file. This is done using the PUT# statement.

We should not confuse this PUT# statement with the PUT statement that is used to perform graphics operations on the screen. The PUT# statement tells BASIC to "put" the contents of the random-file buffer into the disk file itself. Here is the general syntax of the PUT# statement:

PUT # *File Number, Record Number*

The *File Number* parameter specifies the file to which the random file buffer is to be written and the *Record Number* parameter specifies to which record the data will be written. We'll discuss record numbers in more detail soon, but just remember that the random access is accomplished by knowing which record you want to access.

In our example program, though, we haven't specified the record number. In this case, BASIC will default to using the *next* record in the file. BASIC always keeps track of which record number was the last one used. Since we have just created a new file, there aren't any records in it, and the last record number used will be 0. Upon the first PUT# statement, BASIC will automatically increment the last record number to 1. This will work just fine for creating a new file, where each new record will be automatically added following the last one added.

READING SEQUENTIALLY FROM A RANDOM-ACCESS FILE

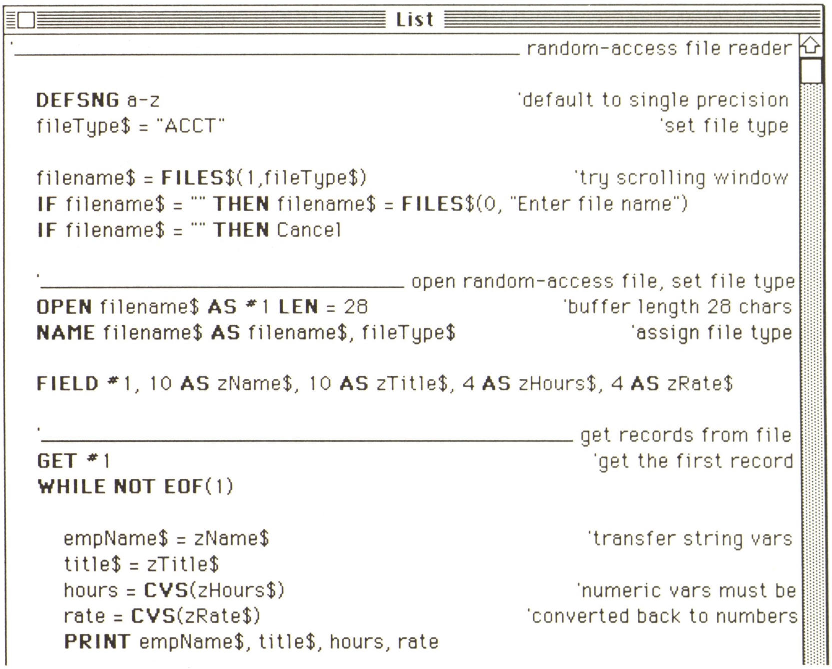
After we have completed creating our new random-access employee file and entered some records into it, we'll no doubt want some way of listing the entire file to see if all the data is correct. This will be the objective of our next example program: to read the data in sequential order from our random-access file.

Even though the file permits *random access,* we can still access the information serially if we want to. This will provide the clearest illustration of the use of the BASIC statements and functions that control access to the file. Later, we'll see how to truly access the file in random order.

Reading data from a random-access file involves a process that is the reverse of the one we used to write the data. There are three steps involved in reading records sequentially from a random-access file:

1. Load a record from the file to the random-file buffer
2. Check for the end-of-file condition
3. Transfer the contents of the buffer variables to program variables

Our next example program, shown in Figure 16-19, will be used during the following discussion. You can skip over the initialization section of the program, since it's identical to the one in our previous program.



```
'____________________________________________________ random-access file reader

  DEFSNG a-z                                          'default to single precision
  fileType$ = "ACCT"                                                 'set file type

  filename$ = FILES$(1,fileType$)                             'try scrolling window
  IF filename$ = "" THEN filename$ = FILES$(0, "Enter file name")
  IF filename$ = "" THEN Cancel

  '________________________________________ open random-access file, set file type
  OPEN filename$ AS #1 LEN = 28                             'buffer length 28 chars
  NAME filename$ AS filename$, fileType$                          'assign file type

  FIELD #1, 10 AS zName$, 10 AS zTitle$, 4 AS zHours$, 4 AS zRate$

  '______________________________________________________________ get records from file
  GET #1                                                      'get the first record
  WHILE NOT EOF(1)

    empName$ = zName$                                         'transfer string vars
    title$ = zTitle$
    hours = CVS(zHours$)                                     'numeric vars must be
    rate = CVS(zRate$)                                     'converted back to numbers
    PRINT empName$, title$, hours, rate
```

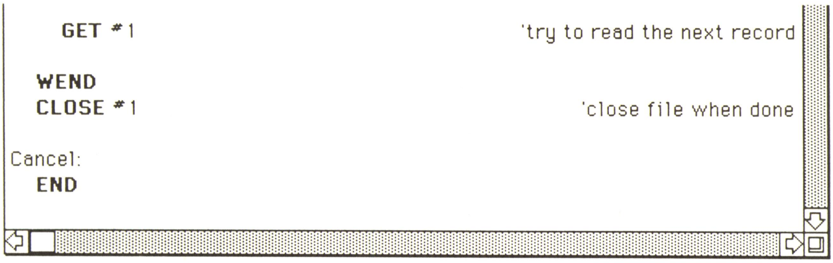
FIGURE 16-19

A program that reads records sequentially from a random-access file

(continued)

FIGURE 16-19

(continued)



```
    GET #1                                       'try to read the next record

  WEND
  CLOSE #1                                       'close file when done

Cancel:
  END
```

Moving data from the file to the buffer: The GET# statement

The first BASIC statement we see in the main file-reading section of our program is a solitary statement: GET #1. This is the BASIC statement that actually reads a record from a random-access file. The general syntax for the GET# statement is:

GET # *File Number, Record Number*

As you can see, the GET# statement uses the same parameters that the PUT# statement uses, except that it does the opposite operation; it "gets" the data from the file into the buffer.

The *Record Number* parameter is how we specify which record we want from the file. In our example program, we've omitted this parameter. BASIC will use the "next" record number to access the file. Since this will be the first time our program tries to get a record from the file, the "next" record is automatically the first record in the file.

Don't confuse this GET# statement with the "screen" GET statement discussed in the next chapter which is used to "get" an area of pixels into an array. They are quite different in purpose, even if their names are similar. The GET# statement uses the field width specifications provided in the FIELD statement to actually read data from the disk file and arrange it in the random-file buffer, where it can be accessed through the buffer variables. More on that in a moment.

Detecting the end of a random-access file: The EOF function

The EOF function is used with random-access files as well as with the sequential-access files we discussed earlier. It works a little differently with random-access files because our program could attempt to access any record number, even if the number was not actually in the file. When used with a random-access file, the EOF function will return a *true* value if the last attempt to access the file (either a GET# or PUT#) specified a record number that would be past the end of the file.

This provides the perfect mechanism for controlling the operation of our program so that it will sequentially read all the records in our file and automatically stop after the last one. To construct a program loop, we use the statement:

```
WHILE NOT EOF(1)
```

Moving data from the buffer to the program: The buffer variables

Inside the WHILE...WEND loop, we find the statements that are required to access the record that is now in the random-file buffer as a result of the GET# statement just above. Remember that strings and numbers are handled differently in random-access files, so in the following sections we'll discuss each of them separately.

String data

The two string fields in our file, the employee name and title, are present in the random-file buffer, accessible through the special buffer variable names, *zName$* and *zTitle$*. It's a simple matter to assign these strings to our own string variables, *empName$* and *title$*.

Converting strings to numeric data: The CVI, CVS, and CVD functions

Remember, when we created our file in the last program we converted the numeric variables *hours* and *rate* to strings before we placed them into the random-file buffer. When we read this data from the file we created back into the random-file buffer, it will be in its string form. Since we want to use the numeric values of these variables in our program, we'll have to convert them back to numeric variables.

There are three functions that convert buffer variables into numeric variables: CVI, CVS, and CVD. These functions are the counterparts of the MKI$, MKS$, and MKD$ functions we used earlier to convert numeric variables to the string form necessary for the random-file buffer. Here are the syntaxes of the three functions:

CVI(*Buffer Variable*) [converts a string to an integer variable]
CVS(*Buffer Variable*) [converts a string to a single-precision variable]
CVD(*Buffer Variable*) [converts a string to a double-precision variable]

Our example program only uses the CVS function because we used the DEFSNG statement at the beginning to define all numeric variables as single-precision.

The last GET#

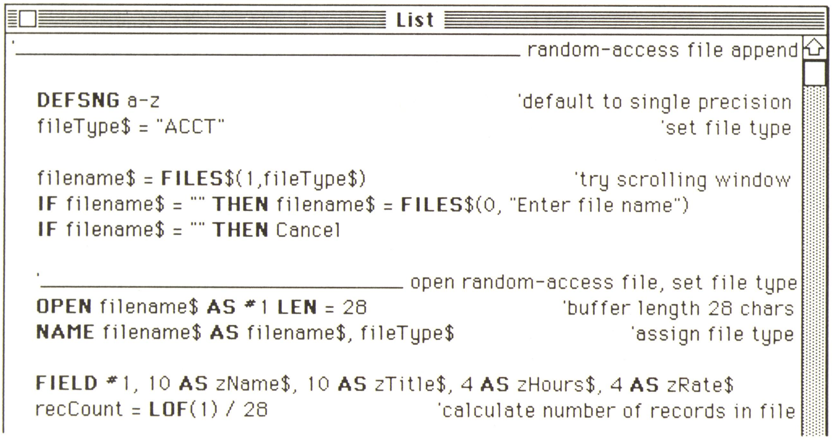
After we display the data with a PRINT statement, we will see another GET# statement that's located just before the end of the WHILE... WEND loop. It is this last GET# statement that keeps the program reading sequentially through the file.

When no *Record Number* parameter is specified, the GET# statement *tries to read* the next record in sequential order. If there is another record, then it will be read into the random-file buffer. But if there are no more records, BASIC sets the EOF function to *true.*

Because the GET# statement is at the end of the loop, the very next thing to happen is that BASIC will see if the condition that keeps the WHILE...WEND loop going is still true. Of course, we've used the NOT function to reverse the logic of the expression in *WHILE NOT EOF(1),* so that the loop operates as long as the EOF function is *not true.*

ADDING DATA TO A RANDOM-ACCESS FILE

Now that we've successfully created a random-access file and listed it sequentially, let's get an idea of what *random access* is all about. One of the most likely things we'll want to do with our new file is to *add new data* to the file. Since it is a random-access file, we'll have to know how many records are already in the file so that we can add the new records at the end. We'll be using a new example program shown in Figure 16-20.



```
'_______________________________________________ random-access file append

  DEFSNG a-z                                    'default to single precision
  fileType$ = "ACCT"                                           'set file type

  filename$ = FILES$(1,fileType$)                       'try scrolling window
  IF filename$ = "" THEN filename$ = FILES$(0, "Enter file name")
  IF filename$ = "" THEN Cancel

  '_____________________________ open random-access file, set file type
  OPEN filename$ AS #1 LEN = 28                        'buffer length 28 chars
  NAME filename$ AS filename$, fileType$                      'assign file type

  FIELD #1, 10 AS zName$, 10 AS zTitle$, 4 AS zHours$, 4 AS zRate$
  recCount = LOF(1) / 28                  'calculate number of records in file
```

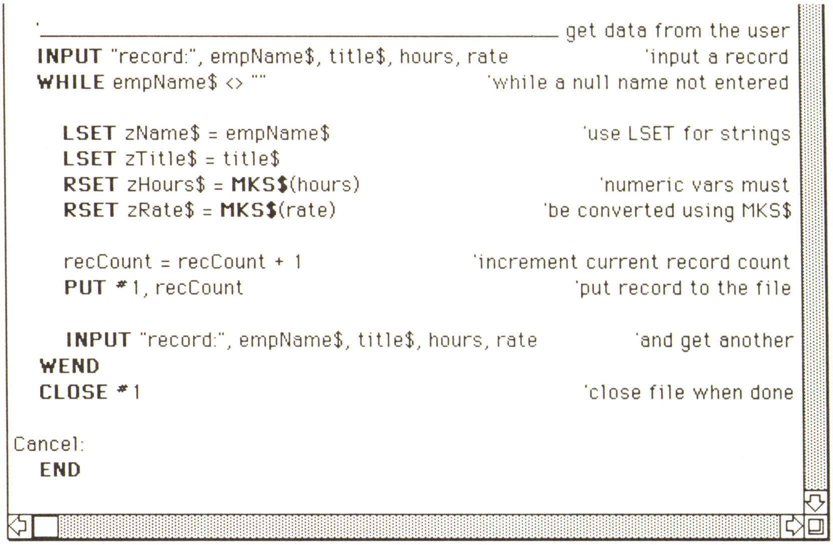
FIGURE 16-20

Adding data to a random-access file

(continued)

FIGURE 16-20

(continued)



```
  '___________________________________________ get data from the user
  INPUT "record:", empName$, title$, hours, rate            'input a record
  WHILE empName$ <> ""                               'while a null name not entered

    LSET zName$ = empName$                                   'use LSET for strings
    LSET zTitle$ = title$
    RSET zHours$ = MKS$(hours)                                 'numeric vars must
    RSET zRate$ = MKS$(rate)                          'be converted using MKS$

    recCount = recCount + 1                       'increment current record count
    PUT #1, recCount                                       'put record to the file

    INPUT "record:", empName$, title$, hours, rate               'and get another
  WEND
  CLOSE #1                                                'close file when done

Cancel:
  END
```

Finding the length of a file: The LOF function

Notice that the initialization section of this program is very similar to the previous two example programs. We have an OPEN statement (with no mode parameter) followed by the FIELD statement which defines the buffer variables.

BASIC provides a simple function to help us determine the length of a file. The LOF function returns the length of the specified file in *bytes.* Since 1 character equals 1 byte, and we know that each record in our file consists of 28 characters, we can calculate the number of records in the file using the statement:

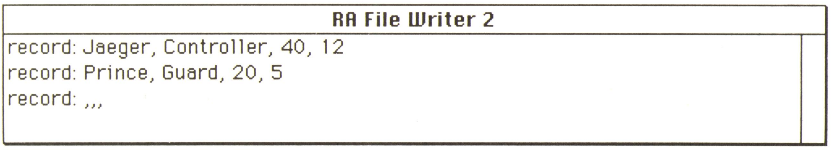
```
recCount = LOF(1) / 28
```

Incrementing the record count

Once we know the number of existing records in our random-access file, we can add new ones. All we have to do is specify the *Record Number* parameter in the PUT# statement that writes the record to the file.

As you can see in Figure 16-20, we are using the same BASIC statements as in the program which first created the random-access file to input the data from the keyboard and set up the buffer variables. The only difference is that we have included the statement:

```
recCount = recCount + 1
```



```
                              RA File Writer 2
record: Jaeger, Controller, 40, 12
record: Prince, Guard, 20, 5
record: ,,,
```

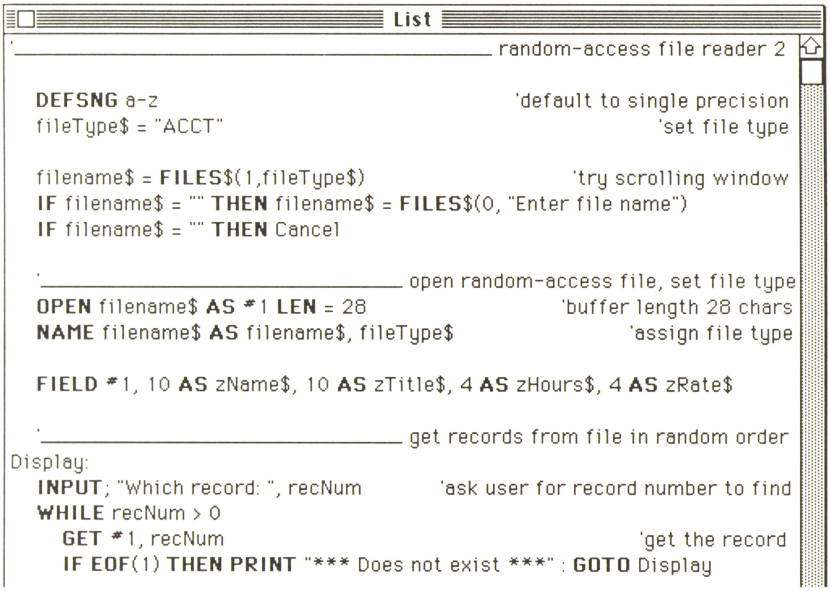
FIGURE 16-21

Two records added to the file

just before the PUT# statement at the end of the loop. This *increments* the variable *recCount* which is then used as the *Record Number* parameter for the following PUT# statement. Each time through the loop, the current record count is incremented so that the new records are successively added to the end of the file. Figure 16-21 shows the screen after running this program. Notice that we've added two employees to the file.

RANDOM-ACCESS BY RECORD NUMBER

We've learned a lot about random-access files in the preceding sections. But we still haven't really used our file in a random fashion. The reason for this is that we wanted to concentrate our attention on the various statements and functions needed to use random-access files. Now that we have that foundation built, let's try accessing our file randomly, modifying RA File Reader to produce the program in Figure 16-22.



```
'______________________________________________ random-access file reader 2

   DEFSNG a-z                                       'default to single precision
   fileType$ = "ACCT"                                              'set file type

   filename$ = FILES$(1,fileType$)                         'try scrolling window
   IF filename$ = "" THEN filename$ = FILES$(0, "Enter file name")
   IF filename$ = "" THEN Cancel

   '__________________________________ open random-access file, set file type
   OPEN filename$ AS #1 LEN = 28                         'buffer length 28 chars
   NAME filename$ AS filename$, fileType$                       'assign file type

   FIELD #1, 10 AS zName$, 10 AS zTitle$, 4 AS zHours$, 4 AS zRate$

   '__________________________________ get records from file in random order
Display:
   INPUT; "Which record: ", recNum         'ask user for record number to find
   WHILE recNum > 0
      GET #1, recNum                                             'get the record
      IF EOF(1) THEN PRINT "*** Does not exist ***" : GOTO Display
```

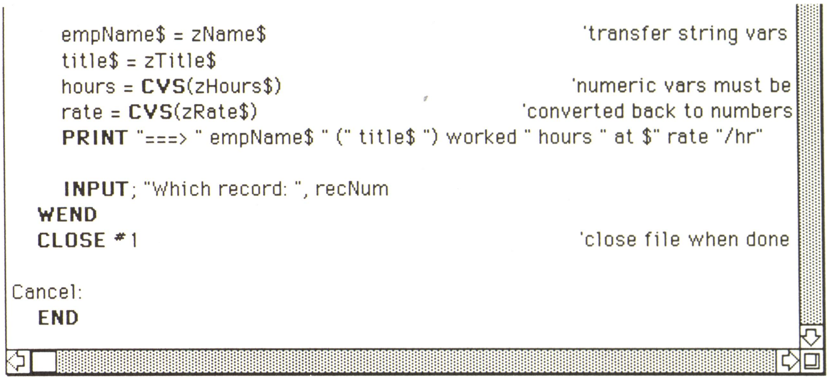
FIGURE 16-22

Random-access by record number

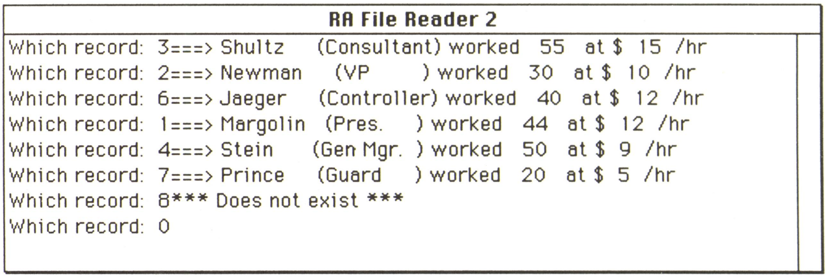
(continued)

FIGURE 16-22

(continued)



```
    empName$ = zName$                                'transfer string vars
    title$ = zTitle$
    hours = CVS(zHours$)                             'numeric vars must be
    rate = CVS(zRate$)                               'converted back to numbers
    PRINT "===> " empName$ " (" title$ ") worked " hours " at $" rate "/hr"

    INPUT; "Which record: ", recNum
  WEND
  CLOSE #1                                           'close file when done

Cancel:
  END
```



RA File Reader 2

```
Which record: 3===> Shultz    (Consultant) worked  55  at $  15 /hr
Which record: 2===> Newman    (VP        ) worked  30  at $  10 /hr
Which record: 6===> Jaeger    (Controller) worked  40  at $  12 /hr
Which record: 1===> Margolin  (Pres.     ) worked  44  at $  12 /hr
Which record: 4===> Stein     (Gen Mgr.  ) worked  50  at $  9 /hr
Which record: 7===> Prince    (Guard     ) worked  20  at $  5 /hr
Which record: 8*** Does not exist ***
Which record: 0
```

The *Display* section of the program in Figure 16-22 is a simple example showing how to access any record in a random-access file. The initialization section is the same as the one in our previous example (although it's not necessary to know the *record count* in this program).

Notice that we're using an INPUT statement to have the user enter the number of the record to be accessed. Just to keep things simple, we'll agree that if the user enters a zero or negative record number, the program will end. The WHILE...WEND loop takes care of this construct. The Output window in Figure 16-22 shows the results of an example user session with this program.

Inside the loop, we use a GET# statement with the record number *recNum* specified. This tells BASIC to try reading the specified record from our file. Remember that if the specified record is beyond the end of the file, the EOF function will become *true*. This allows us to use an IF...THEN statement to produce an "error message" if an invalid record number is entered by the user. The rest of the program is self-evident. We see the same use of the buffer variables that we've seen throughout this chapter. Since we're reading the data from the file, we need to convert the numeric fields from their string-like buffer variables into actual numeric variables using the CVS function.

USING AN INDEX TABLE WITH A RANDOM-ACCESS FILE

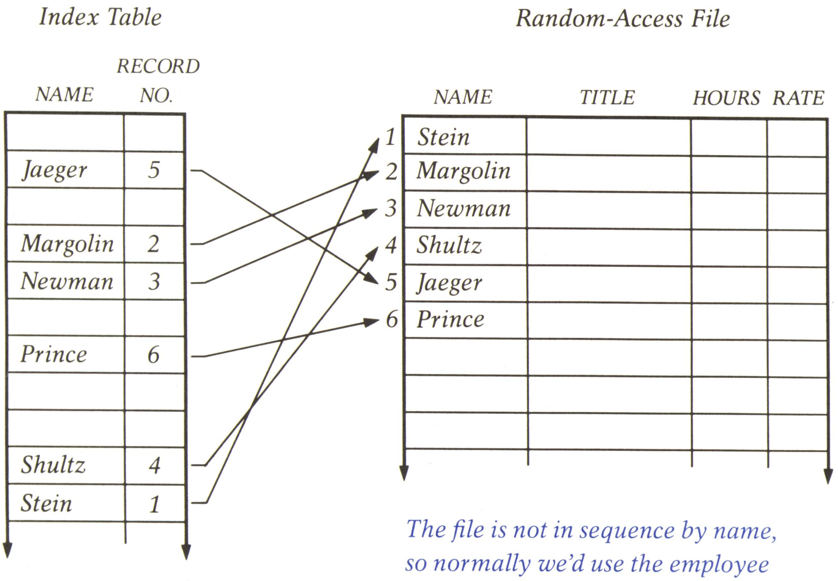
In our previous example, we learned that we can specify a particular record number, and BASIC will return the data from that record, and put it into the random-file buffer. That's great if we know which record contains the data we're looking for. For example, suppose we wanted to know how many hours employee Shultz worked. We would have to know in advance that Shultz's data was located in record number 4 in the file.

One way this can be done is to assign each employee an "employee number" that's the same as the record number in the Employee File where their data is located. We could get then Shultz's data by specifying Shultz's employee number as the record number. But this would be one more number the user would have to keep track of, and one more number the user could enter incorrectly.

There is a more intelligent way to use random-access files, but it involves a lot more work for the programmer. Ideally, we should be able to enter an employee's name, and have our program find their data record in the file. To do this, we use an *index table* as shown in Figure 16-23.

FIGURE 16-23

Using an index table to access a random-access file


Index Table
Random-Access File
NAME
RECORD NO.
Jaeger 5
Margolin 2
Newman 3
Prince 6
Shultz 4
Stein 1
NAME
TITLE
HOURS
RATE
1 Stein
2 Margolin
3 Newman
4 Shultz
5 Jaeger
6 Prince

The index table is sorted alphabetically, and contains the record number as a pointer *to the file*

The file is not in sequence by name, so normally we'd use the employee number as the RECORD NO.

The exact nature of this process is beyond the scope of this book. Therefore, we're only going to diagram the concept here, and not develop a BASIC program to do this.

The general concept involves having an index table containing the employee names, sorted alphabetically, and the record number where their data is located in the file. This index table can take the form of an array that is set up at the beginning of the program by sequentially reading through the entire file once.

Before we can look up an employee's name in the index table, it must be *sorted* into ascending alphabetic sequence by the employee names. There are several methods of sorting data in arrays, depending upon the size of the array to be sorted.

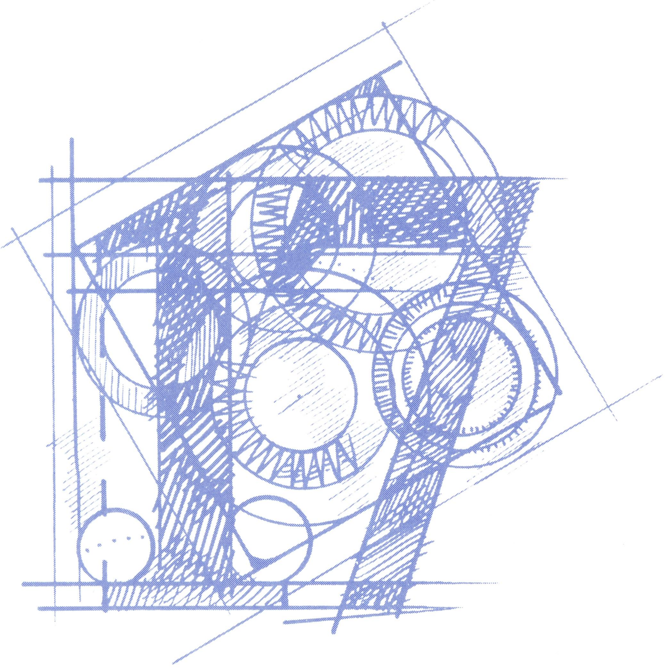
Once the index-table array has been sorted alphabetically, we can have our program "search" for a particular employee name. If the name is found in the index table, then we can simply use the record number that is associated with that name to access the data in the file. If the name is not found in the index table, then we know that there is no corresponding data record in the file.

The index-table method of accessing a random-access file can be used in many different ways depending upon the needs of the user. Also, there are a number of variations on this approach, depending on the size and complexity of the data file.

REFRESH-YOURSELF WINDOW

In this chapter, we've covered one of the most powerful features of Microsoft BASIC for the Macintosh. The ability to use disk files to handle large amounts of data, and to save it for future use, gives our programs a true sense of intelligence. With these tools, we can begin to write BASIC programs that really do help us in our lives. We can save any kind of information that might be of interest to us at a later time, and then simply open a file to read it back into our program.

But we've been concentrating for a long time on a very difficult subject, which is not always too interesting. If you've been a good student of BASIC, and have methodically worked your way through our study of files, then you have a reward coming. In our next chapter, we're going to show you how to create *animation* with the graphics statements that are available in BASIC.



Animation

The word animation *has a variety of meanings, ranging from “movie cartoons” to “talking excitedly while waving your hands.” In this chapter we’re going to use* animation *to mean “images that move about on the Macintosh screen.” Such images might be, for example, the space ships in a “lunar lander” game, graphs of sales figures that you can alter dynamically with* the mouse, or a representation of a machine whose gears and pistons actually move on the computer screen.

As we'll see, such moving images can be simple ones generated by our program, or complex works of art generated in MacPaint. In this chapter we'll show you how to generate such images, and how to move, enlarge, and "zoom" them. We'll also show you how to give images a dynamic quality by alternating rapidly between multiple views of an object.

Clearly, this chapter cannot tell you everything there is to know about computer animation. In fact, entire books have been devoted to the topic. But what we can do is show you some of the fundamental techniques available for producing animation with Microsoft BASIC for the Macintosh. Once you've learned these techniques, you should be able to apply them to your own animation ideas. Who knows—perhaps you will emerge as the Walt Disney of the computer world.

Since it is impossible to show the effect of motion in the figures in a book, we urge you to type in the example programs in this chapter and run them. In many cases this is the only way to completely understand what the program really does.

THE GET AND PUT STATEMENTS

The most important statements for animation in BASIC are GET and PUT. What do these statements do? To answer this question, let's consider for a moment the central problem posed by animation. The problem arises from the enormous amount of information contained in a graphics image, and from the fact that computers, while they can process data rapidly, are operating at their limits when they attempt to move complex images around the screen. To make animation more practical, we need to speed up and simplify the process of drawing an image on the screen. The GET and PUT statements were designed to do exactly that.

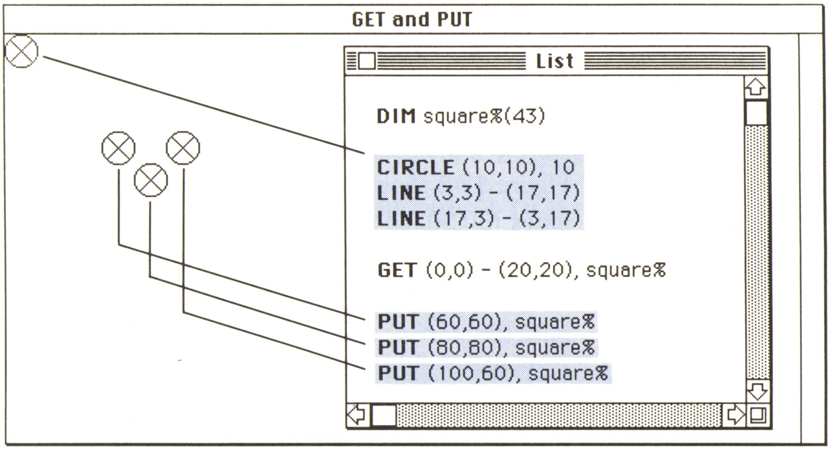
In a nutshell, GET takes an image that is already on the screen and writes it into an array. From the standpoint of the array, it *gets* the image from the screen. PUT does just the opposite. It takes an image from an array and *puts* it on the screen.

How do these operations make animation work better? Imagine that a long and complex series of BASIC statements has been used to generate a picture on the screen. Perhaps we've created the image of an airplane by using a dozen or so LINE and CIRCLE statements. Now we want to make the airplane appear to move by drawing it in a series of new locations across the screen. We could simply repeat our dozen statements, using slightly different parameters, so the airplane would be re-created, line by line, in each new location. However, this would take a long time, and instead of flying smoothly the airplane would move jerkily across the screen.

What we would like is a way to condense the dozen statements which draw the airplane down into a single "magic" statement. Then, whenever we wanted to generate a new image of the airplane in a different position, we'd execute this new statement with the appropriate parameters, and presto, there would be our airplane. Since the single statement would be faster to execute, the airplane would move smoothly. This single "magic" statement is PUT. Its companion statement, GET, is used to store the original image so that PUT can make use of it.

Let's see how this works in a very simple program. The program in Figure 17-1 creates a simple image, then reproduces it three times in three different locations using PUT.

The DIM statement first sets aside storage space in an array, *square%*, large enough to hold the image we are going to create. (We will return later to the question of how to determine the size of the array.) Next, we generate the image using the CIRCLE and two LINE statements. This image—a circle with an *X* in it—appears in the upper left corner of the screen.



```
DIM square%(43)

CIRCLE (10,10), 10
LINE (3,3) - (17,17)
LINE (17,3) - (3,17)

GET (0,0) - (20,20), square%

PUT (60,60), square%
PUT (80,80), square%
PUT (100,60), square%
```

FIGURE 17-1

Using PUT to generate multiple images

The GET statement

At this point, the GET statement makes its appearance. Its purpose is to transfer the image from the screen into the array *square%*. Its format is similar to that of LINE when LINE is used to generate a box. The numbers in the first pair of parentheses are the (x,y) coordinates of the upper left corner of the image. The numbers in the second pair of parentheses are the (x,y) coordinates of the lower right corner. Finally, the name of the array, *square%,* is used to tell GET what array the image should be transferred to.

The PUT statement

Following execution of the GET statement, the image is safely stored in the array. How do we know this? We could, if we wanted, examine each cell of the array individually by executing statements such as *PRINT array%(3)* in the Command window. However, the resulting numbers would be difficult for us to understand. It is far easier to execute a PUT, and see if the image we get back on the screen is the same one we put into the array. Our program, in fact, executes three PUT statements to demonstrate how one image can be reproduced in different places.

The format of PUT is similar to that of GET, except that with PUT we need to give only *one* set of coordinates, since PUT can figure out how big the image is from the size of the array. The statements in parentheses following each PUT are the (x,y) coordinates of the place where we want the *upper left* corner of the image to go; and the name of the array, *square%,* is used to tell PUT which array we want the image to come from. As you can see from the figure, each of the three PUT statements results in the image being reproduced in a different location on the screen.

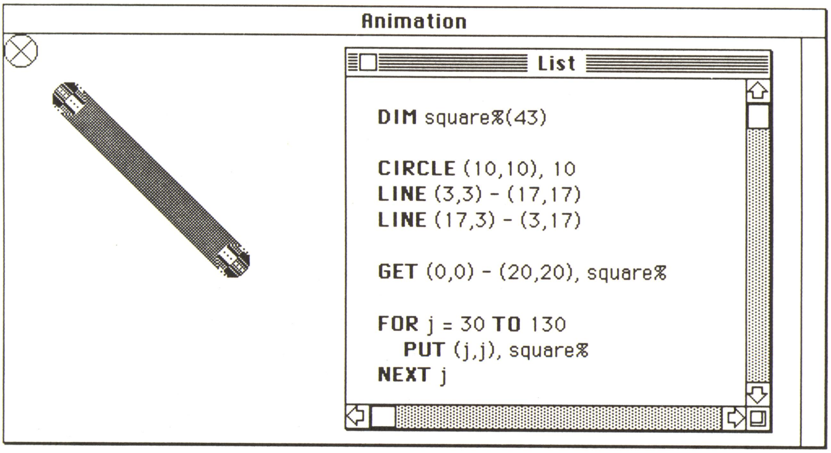
MOVING THE IMAGE

We've said that animation is the creation of images that move. Let's see if we can get our simple "X-in-a-circle" pattern to move across the screen. Figure 17-2 shows a simple program that moves the image from coordinates (30,30) to (130,130).

The program works by executing the PUT statement over and over inside a FOR...NEXT loop. The loop counter variable, *j,* is used as both the starting x and y coordinates in the PUT statement, with the effect that the image is drawn repeatedly along a diagonal line. The effect is much like playing with a rubber stamp and pressing many images in quick succession on the page. Unfortunately, as you can see from the figure, this

FIGURE 17-2

Blurred motion using PUT

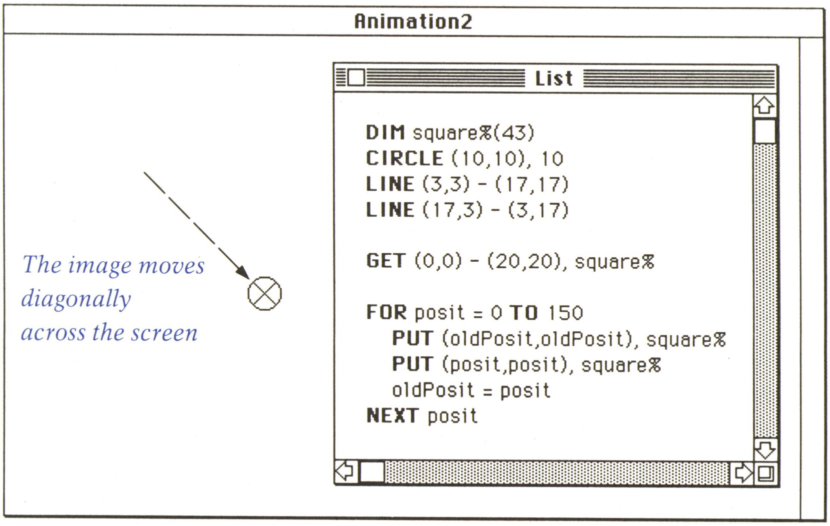


```
DIM square%(43)

CIRCLE (10,10), 10
LINE (3,3) - (17,17)
LINE (17,3) - (3,17)

GET (0,0) - (20,20), square%

FOR j = 30 TO 130
   PUT (j,j), square%
NEXT j
```

program suffers from a serious problem: Because the old images aren't erased, the result is a blur that remains on the screen, rather than a single object that appears to move.

Erasing the old image

Let's try to remedy the blurring problem shown in our last program. We'll do this using a fundamental animation technique: erasing the old image just before we draw the new one. Figure 17-3 shows how this is done.



```
DIM square%(43)
CIRCLE (10,10), 10
LINE (3,3) - (17,17)
LINE (17,3) - (3,17)

GET (0,0) - (20,20), square%

FOR posit = 0 TO 150
   PUT (oldPosit,oldPosit), square%
   PUT (posit,posit), square%
   oldPosit = posit
NEXT posit
```

FIGURE 17-3

Erasing the old image

The principle employed here is this: Each time we draw an image, we also store enough information about its coordinates so that our program can remember where it was. In this case we need to remember only one coordinate, stored in the variable *posit.* We reassign its contents to a variable called *oldPosit,* just after we've drawn the image on the screen with the statement *PUT (posit,posit), square%.*

Each time through the loop the image is drawn and its coordinates are stored in the variable *oldPosit.* The next time through the loop the variable *posit* will change: In our program it will increase by 1, since it is the counter variable in the loop. However, the old value of *posit* will be stored in *oldPosit.* Now, when the statement *PUT (oldPosit,oldPosit), square%,* is executed, it will draw a new image *right on top of the old one.* This will cause the old image to disappear. Why? The answer to this question is the subject of the next section but it might help you to know now that PUT uses the same kind of XOR graphics we studied in Chapter 15 on QuickDraw commands. The effect is that if you draw an image exactly on top of another image using a normal PUT statement, the two images will "cancel out" and disappear. This provides us with a valuable technique for erasing an image when we don't need it any more.

Now that the old image has been erased in our program, the statement *PUT (posit,posit), square%* creates another image at the new *posit* coordinates. By repeatedly drawing an image, then erasing it and redrawing it at a new location, we create the illusion of smooth motion, as you will see when you run the program.

INTERACTION WITH THE BACKGROUND

We've already seen how to erase an image by superimposing the same image on top of it. This is a particular instance of a more general situation. In this section we're going to explore what happens when one image is superimposed on another in a variety of different ways.

So far we've moved our image around on the screen using the program itself to generate new coordinates in a FOR...NEXT loop. Let's see if we can get closer control of our image. Figure 17-4 lists a program that will connect our "X-in-a-circle" image to the mouse pointer.

As you can see, the program works by reading the (x,y) coordinates of the mouse pointer, and uses PUT to draw the image at these new coordinates. Actually, the program subtracts 20 from the mouse coordi-

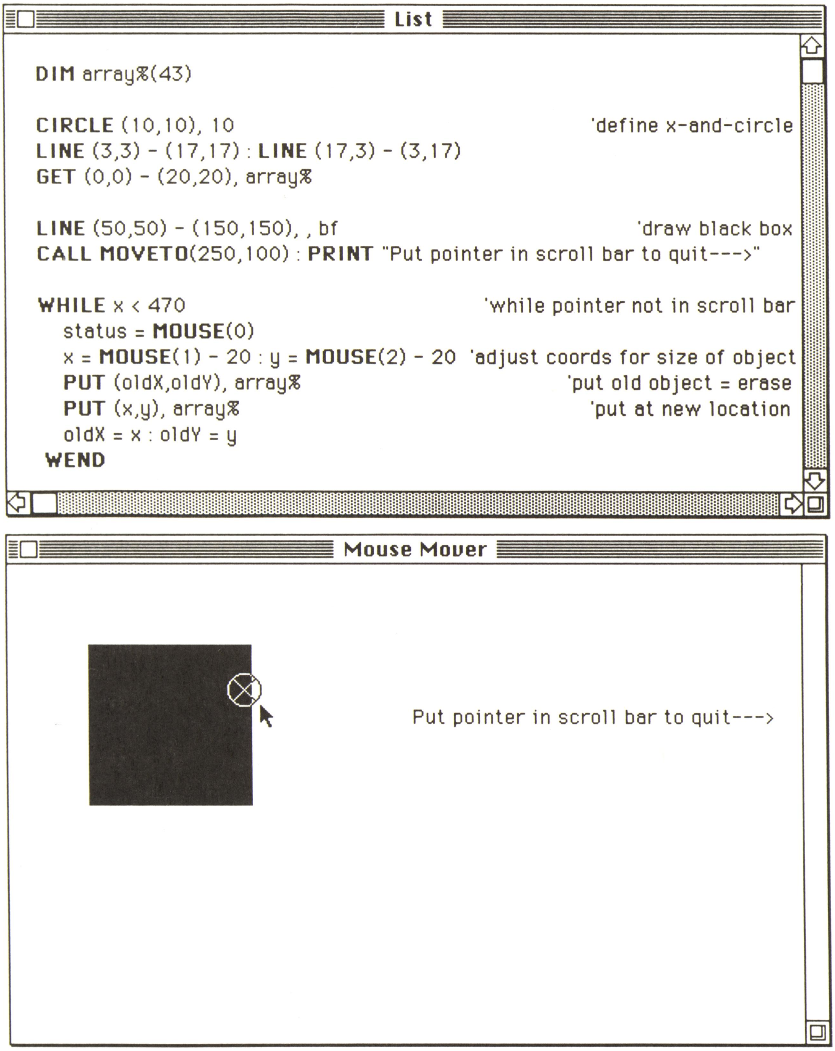


FIGURE 17-4

The Mouse-Mover program

```
DIM array%(43)

CIRCLE (10,10), 10                                   'define x-and-circle
LINE (3,3) - (17,17) : LINE (17,3) - (3,17)
GET (0,0) - (20,20), array%

LINE (50,50) - (150,150), , bf                              'draw black box
CALL MOVETO(250,100) : PRINT "Put pointer in scroll bar to quit--->"

WHILE x < 470                               'while pointer not in scroll bar
  status = MOUSE(0)
  x = MOUSE(1) - 20 : y = MOUSE(2) - 20  'adjust coords for size of object
  PUT (oldX,oldY), array%                           'put old object = erase
  PUT (x,y), array%                                   'put at new location
  oldX = x : oldY = y
 WEND
```

```
Put pointer in scroll bar to quit--->
```

nates, so that the pointer points to the lower right corner of the image, rather than the upper left corner as the PUT statement normally would.

We've also drawn a black box and a line of text on the screen. This gives us the opportunity to see how the image interacts with different backgrounds. Once you've typed in the program, try moving the image around the screen. You'll see that it appears black on a white background, and white on a black background. Also, it doesn't change the background that it passes over, as you can see in the Output window in Figure 17-4.

The XOR action verb

What we've seen is only one way an image can interact with the background. It is, in fact, the default mode that BASIC uses if nothing else is specified. To use other modes, we add something called an *action verb* to the PUT statement. The default action verb is XOR. Therefore, the two PUT statements:

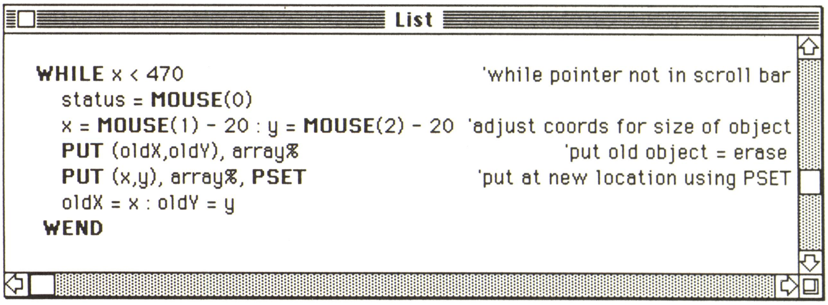
```
PUT (x,y), square%
PUT (x,y), square%, XOR
```

are exactly equivalent.

What does XOR mean in practice? As you learned in Chapter 15, "QuickDraw," XOR means that if *either* the foreground image or the background is black, the resulting image will be black, but if they are *both* black, the resulting image will be white. In many cases this is exactly what we want. However, there are times when other action verbs can be useful. Let's look at them briefly.

The PSET action verb

There may be times when you want your image to *erase* the background as it passes over it. Perhaps, for example, you're writing a game in which you want chunks of an enemy fort (which is black) removed by gunfire. PSET will do it. Notice, however, that when an image is superimposed on a black background, the background will be removed *behind the entire array-rectangle holding the image,* not just behind the image itself. Figure 17-5 shows how this looks, using our Mouse-Mover program where both the rectangle and the text have been altered. The only change to the program is the addition of the word PSET following the PUT statement.



```
WHILE x < 470                                    'while pointer not in scroll bar
  status = MOUSE(0)
  x = MOUSE(1) - 20 : y = MOUSE(2) - 20  'adjust coords for size of object
  PUT (oldX,oldY), array%                          'put old object = erase
  PUT (x,y), array%, PSET                     'put at new location using PSET
  oldX = x : oldY = y
WEND
```

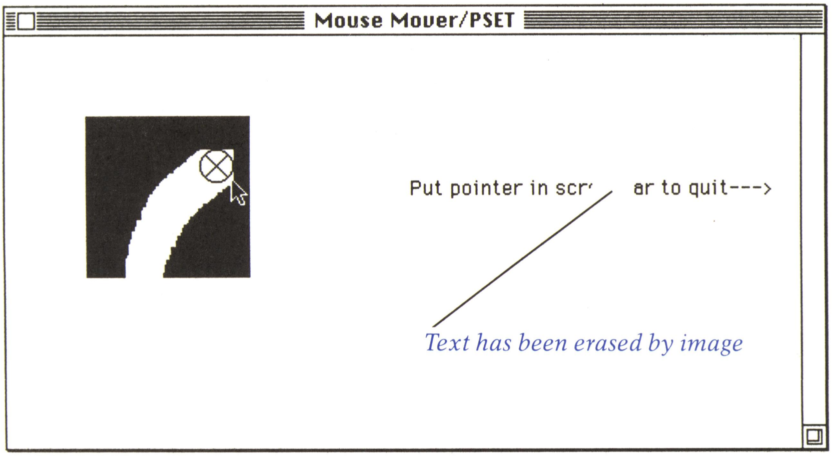
FIGURE 17-5

Demonstrating the PSET action verb

(continued)

FIGURE 17-5

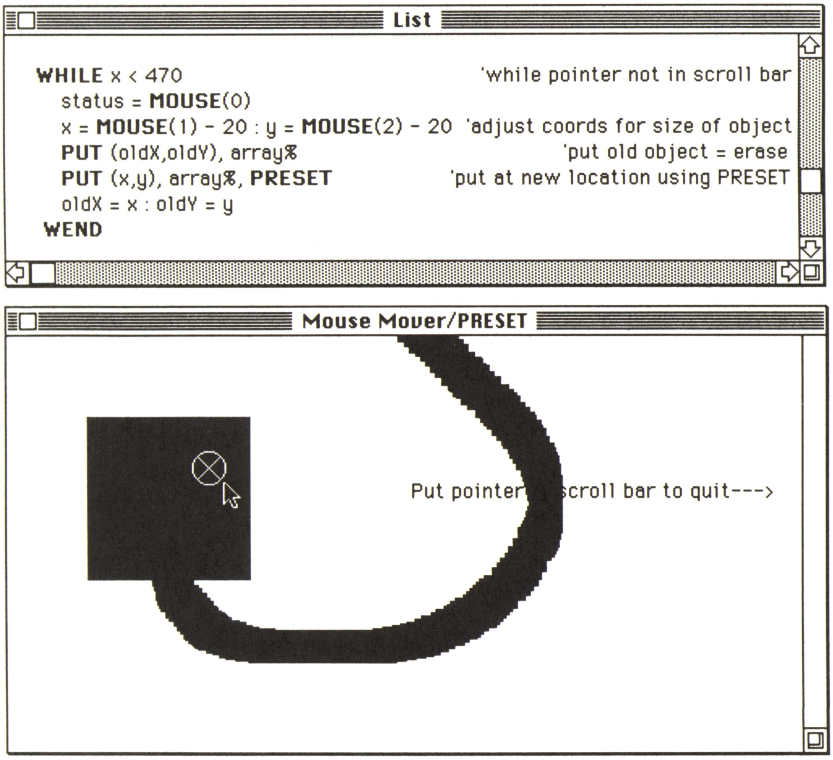
(continued)


Mouse Mover/PSET
Put pointer in scr ar to quit--->

Text has been erased by image

The PRESET action verb

PRESET can be thought of as the opposite of PSET. Instead of erasing (changing to white) everything it passes over, it turns it to black, whether it was white or black to begin with. This might be useful for drawing boxes or lines in the trail of a moving image. Figure 17-6 shows how this looks.


List
Mouse Mover/PRESET
Put pointer scroll bar to quit--->

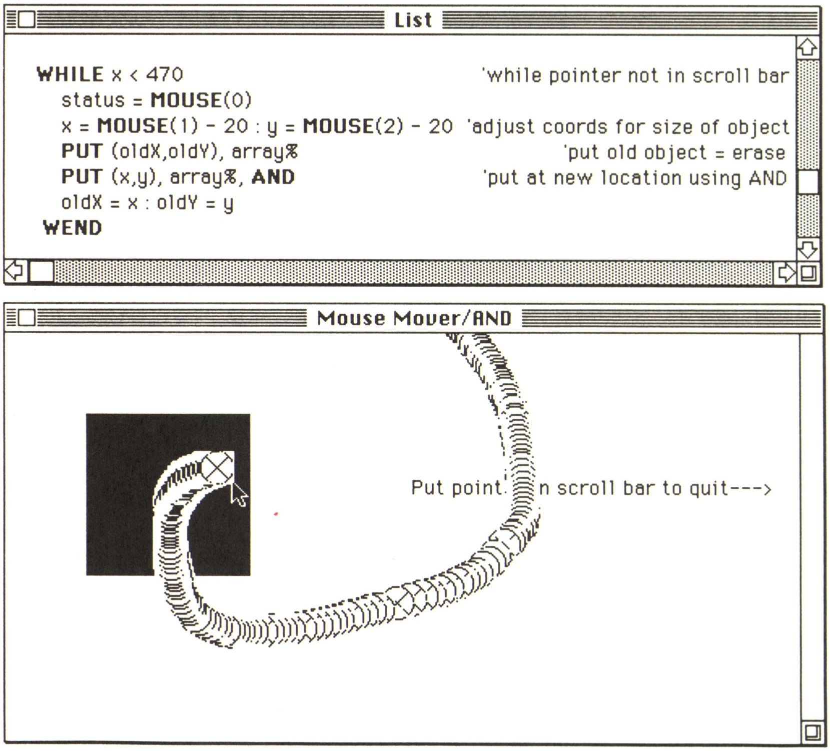
```
WHILE x < 470                                 'while pointer not in scroll bar
  status = MOUSE(0)
  x = MOUSE(1) - 20 : y = MOUSE(2) - 20  'adjust coords for size of object
  PUT (oldX,oldY), array%                         'put old object = erase
  PUT (x,y), array%, PRESET                 'put at new location using PRESET
  oldX = x : oldY = y
 WEND
```

FIGURE 17-6

Demonstrating the PRESET action verb

The AND action verb

As its name implies, AND creates a black pixel only when both the foreground image *and* the resulting image are both black. Thus an image drawn on a black background will be visible, but one drawn on white will not. This might be useful if you want to show images that seem to disappear "behind" white objects. AND produces somewhat unpredictable results with moving images, however, as can be seen in Figure 17-7 where it interacts with the previous PUT in XOR mode.



```
WHILE x < 470                                    'while pointer not in scroll bar
   status = MOUSE(0)
   x = MOUSE(1) - 20 : y = MOUSE(2) - 20  'adjust coords for size of object
   PUT (oldX,oldY), array%                              'put old object = erase
   PUT (x,y), array%, AND                          'put at new location using AND
   oldX = x : oldY = y
 WEND
```

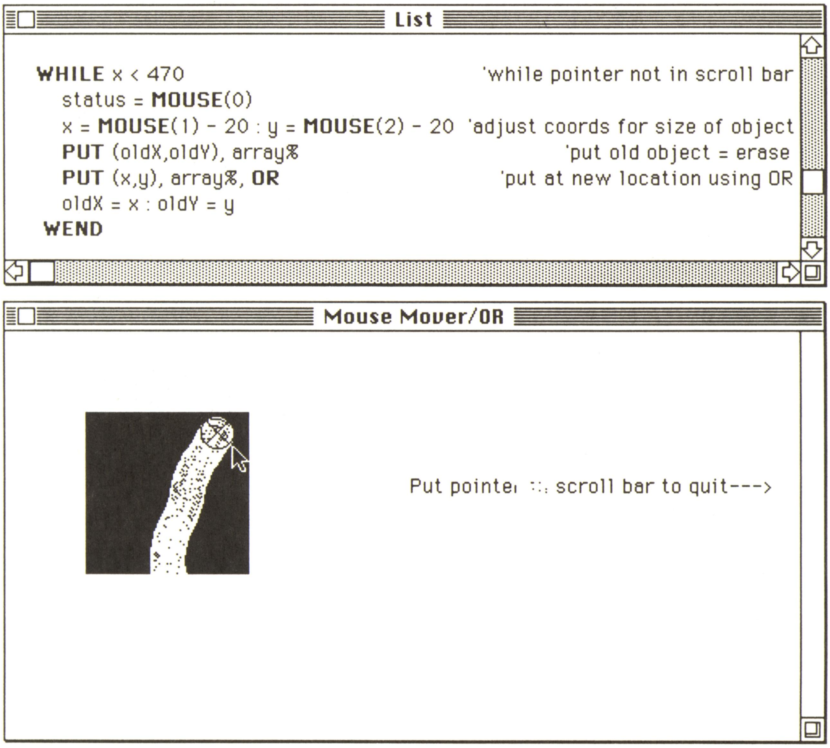
Put point n scroll bar to quit--->

FIGURE 17-7

Demonstrating the AND action verb

The OR action verb

Earlier we saw that, with XOR, if either the foreground image or the background image is black, then the resulting image will be black. OR differs from XOR in that if either the foreground or the background image, *or both,* is black, then the new background will be black. The result, in terms of our program as shown in Figure 17-8, is that an *animated* image can be seen against both black and white backgrounds, but that—unlike using XOR alone—it destroys the image it passes over. An *unanimated* image will only show up against a white background, so OR could be used in another program to make images appear to vanish when passing "behind" something black.



```
WHILE x < 470                                       'while pointer not in scroll bar
   status = MOUSE(0)
   x = MOUSE(1) - 20 : y = MOUSE(2) - 20  'adjust coords for size of object
   PUT (oldX,oldY), array%                                  'put old object = erase
   PUT (x,y), array%, OR                              'put at new location using OR
   oldX = x : oldY = y
 WEND
```

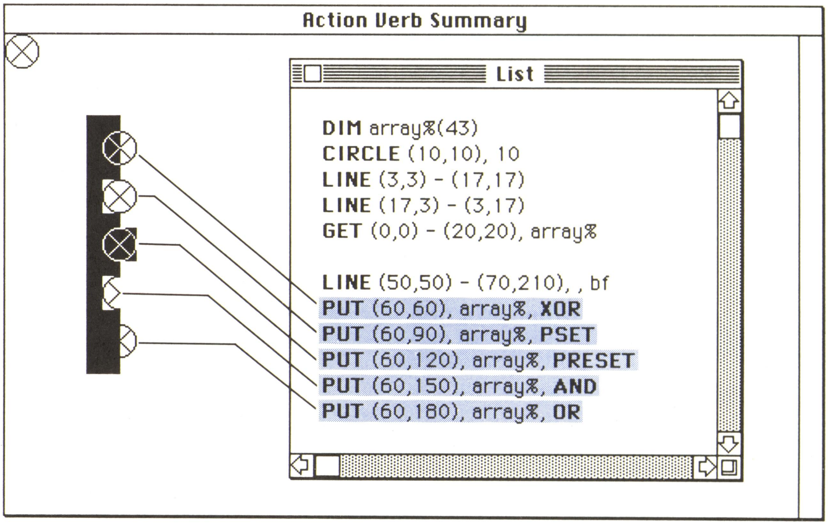
```
Put pointer in scroll bar to quit--->
```

FIGURE 17-8

Demonstrating the OR action verb

FIGURE 17-9

Action verb summary



```
DIM array%(43)
CIRCLE (10,10), 10
LINE (3,3) - (17,17)
LINE (17,3) - (3,17)
GET (0,0) - (20,20), array%

LINE (50,50) - (70,210), , bf
PUT (60,60), array%, XOR
PUT (60,90), array%, PSET
PUT (60,120), array%, PRESET
PUT (60,150), array%, AND
PUT (60,180), array%, OR
```

Figure 17-9 summarizes how all the action verbs look when the image is not moved against the background. You'll probably need to experiment to determine which of the action verbs is appropriate for a particular animation project.

CALCULATING THE SIZE OF THE ARRAY

Earlier we said that we would defer until later explaining how we knew to make the *square%* array in our first program 43 integers long. Now it's time to bite the bullet and figure this out. The Microsoft BASIC manual shows a rather complicated formula for calculating the array size. We can simplify this somewhat by assuming that the image will always be in the upper left corner of the screen, which is the case in all the examples in this chapter. Thus the upper left corner of our image is at (0,0). If the lower right corner of the image is at (x,y), then the simplified formula for the integer array size, expressed in BASIC, is:

$$arraysize = \mathbf{INT}(\mathbf{1} + x / \mathbf{16}) * (y + \mathbf{1}) + \mathbf{2}$$

That's a pretty intimidating formula, but let's plug the numbers in for our first program and see what happens. Don't worry if algebra is Greek to you; in a moment we'll show you how to get your computer to do what it does best: solve the formula for you.

Refer back to the “X-in-a-circle” image of Figure 17-1. We need a 20- by 20-pixel array to hold this image, since the circle’s diameter is 20 pixels (a radius of 10). Thus, in the above formula, both x and y are 20. Plugging these into the equation gives:

$$\mathit{arraysize} = \mathbf{INT(1 + 20 / 16) * (20 + 1) + 2}$$
$$\mathit{arraysize} = \mathbf{INT(2.25) * (21) + 2}$$
$$\mathit{arraysize} = \mathbf{2 * 21 + 2}$$
$$\mathit{arraysize} = \mathbf{44}$$

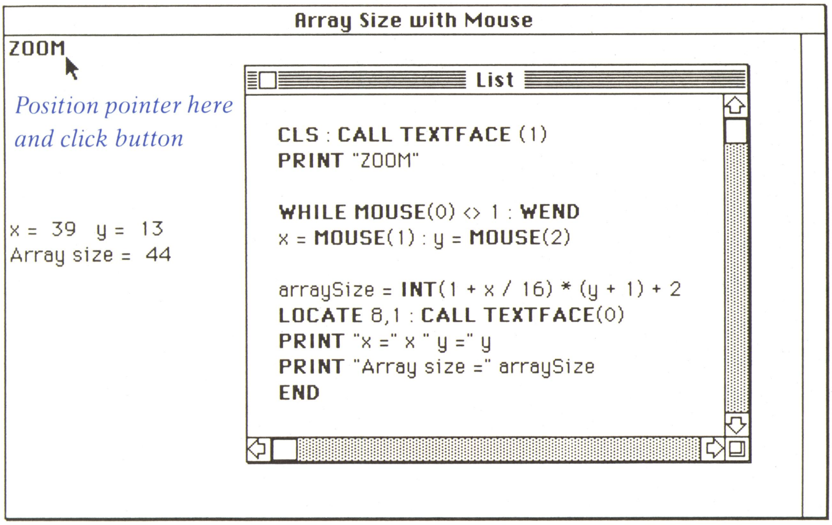
Since the first array element is 0, using 43 for the size of the array gives us the necessary 44 elements.

You should note that this formula differs from the one in the Microsoft manual in that we are assuming the use of *integer arrays,* that is, arrays made of integer variables. The formula in the book tells how many bytes are needed for each data type; and since there are two bytes per integer array element, our integer array need be only half as large as the formula in the book suggests. In this case, 44 integers does it.

Using the mouse to determine array size

But why go to all the trouble to figure out the array size ourselves? After all, BASIC and the Macintosh are great at just this sort of thing. Figure 17-10 shows a program that uses the mouse to calculate the array size needed for a particular image.

What image is the program starting with? In the case of this particular program, the image is the word *ZOOM,* formed on the screen with a



```
ZOOM
```

Position pointer here and click button

```
x = 39  y = 13
Array size =  44
```

```
CLS : CALL TEXTFACE (1)
PRINT "ZOOM"

WHILE MOUSE(0) <> 1 : WEND
x = MOUSE(1) : y = MOUSE(2)

arraySize = INT(1 + x / 16) * (y + 1) + 2
LOCATE 8,1 : CALL TEXTFACE(0)
PRINT "x =" x " y =" y
PRINT "Array size =" arraySize
END
```

FIGURE 17-10

Using the mouse to calculate array size

PRINT statement. We've done this partly to show you that any image on the screen can be used in GET and PUT statements, even if it was created by PRINT rather than a graphics statement. However, the section of code starting with *WHILE MOUSE(0) <> 1* down to *END* can be taken out of this program and substituted in any program you have written that draws an image in the upper left corner. These lines will give you the array size, and you can then remove them from your program.

When we run the program, the image is generated on the screen and then the program goes into a loop, waiting for us to click the mouse button. Next, we move the mouse pointer to just outside the lower right corner of the image, as shown in Figure 17-10. Then, without moving the pointer, we click the mouse button. The program then prints out the (x,y) coordinates of the lower right corner of the specified rectangle. We can then use these coordinates in GET statements in our program. This program also prints out the number of array elements. In this example, as in the previous one, the array size is 44 elements.

Later in this chapter we'll show you a more sophisticated version of this program, which we'll use to transfer images from MacPaint.

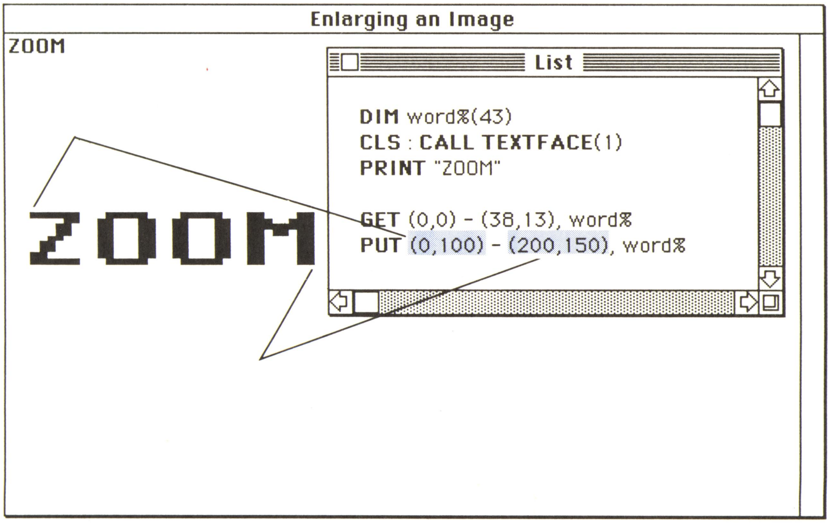
CHANGING THE IMAGE SIZE

In the last program we printed the word *ZOOM* on the screen. The use of this word was not accidental. We're going to take this image and show you how it can be blown up, much as the zoom lens on a still camera can be used to generate a larger image. The program that does this is shown in Figure 17-11.

This program first prints the word *ZOOM* in the upper left corner of the screen, using the bold textface. The GET statement then puts the image into an array called *word%*. Now comes the fancy part. It turns out that PUT can specify two sets of coordinates. If only one set is specified, as we've shown so far, the image will be reproduced the same size as the original. However, if two sets of coordinates are used, the image will be *automatically expanded* (or reduced) to fit in the rectangle specified by these coordinates. This is really an amazing business. Think of the work the computer must be doing to calculate where all the pixels should go when the image is, say, 37 percent bigger than it was before. Without this option,

FIGURE 17-11

Enlarging an image



```
DIM word%(43)
CLS : CALL TEXTFACE(1)
PRINT "ZOOM"

GET (0,0) - (38,13), word%
PUT (0,100) - (200,150), word%
```

it would take days to write a program to do something like this, but here's Microsoft, making it as easy as possible for us with the PUT statement.

In our program, we specify that we want the image expanded from 38 by 13 pixels to 200 by 50 pixels. (That is, 200 minus 0 in the x direction, and 150 minus 100 in the y direction.) Notice that this not only changes the size of the resulting image, but the shape of the letters as well: The letters are proportionately wider in the enlarged image.

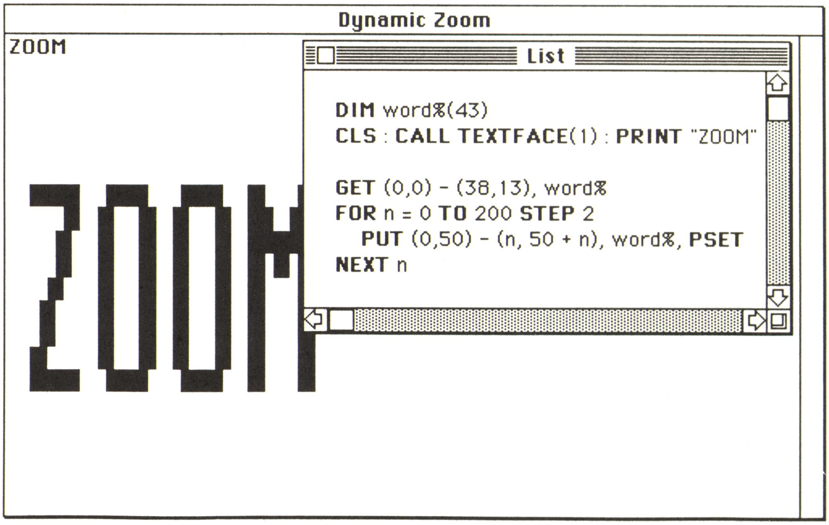
DYNAMIC ZOOMING

We've seen how to enlarge an image. What about having it zoom out at us, changing shape before our very eyes, like a zoom shot in the movies? Try out the program shown in Figure 17-12 on the following page, which creates exactly this effect.

Like the last program, the one in Figure 17-12 starts with the word *ZOOM* as the image, and uses a GET to store it in the array *word%*. However, instead of executing a single PUT, this program nests the PUT in a FOR...NEXT loop, and uses the loop counter variable *n* to increase the size of the image each time the loop is executed. The upper left corner of the image is fixed at (0,50); but the lower right corner starts at (0,50) (when *n* is 0), and moves down and to the right to (200,250) (when *n* is 200). The final size of the image is shown in the Output window of Figure

FIGURE 17-12

A program that demonstrates dynamic zooming



```
DIM word%(43)
CLS : CALL TEXTFACE(1) : PRINT "ZOOM"

GET (0,0) - (38,13), word%
FOR n = 0 TO 200 STEP 2
   PUT (0,50) - (n, 50 + n), word%, PSET
NEXT n
```

17-12. What the figure can't show is the smooth and rapid expansion of the image as the program runs.

Something to notice in this program is that we have used the action verb PSET in the PUT statement. As you recall, PSET erases anything under the image rectangle. This is useful here because each new, larger image erases the old, smaller image underneath it. Without this action verb, the new and old images interact in a complicated way to produce a chaotic and unreadable result. Try using the default XOR, for instance, and see the difference for yourself.

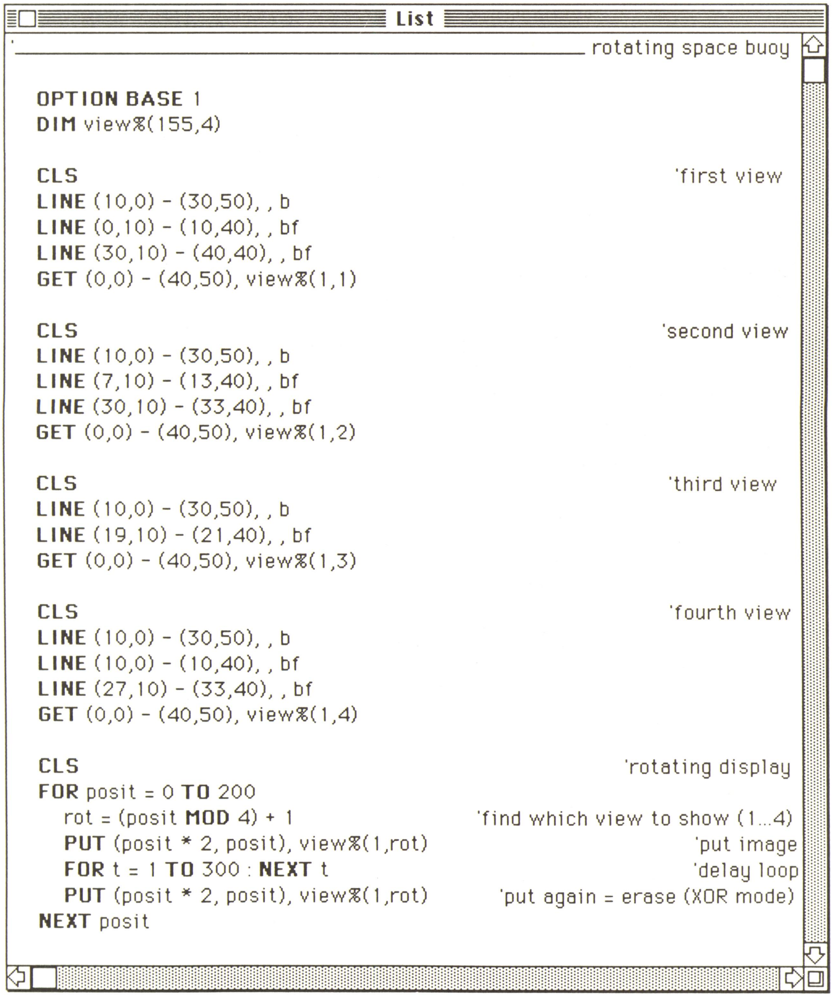
MULTIPLE IMAGES

We've seen how to move images, and how to change their size and even distort their shapes, using the PUT statement. However, these techniques are inadequate in a variety of important animation situations. We can make an airplane fly, but suppose we want its propeller to rotate? We can make an image appear to come toward us, but suppose we want it to turn at the same time? Suppose we want a little man to run across the screen, and we want his legs to move? All these problems can be solved using the technique of *multiple images.*

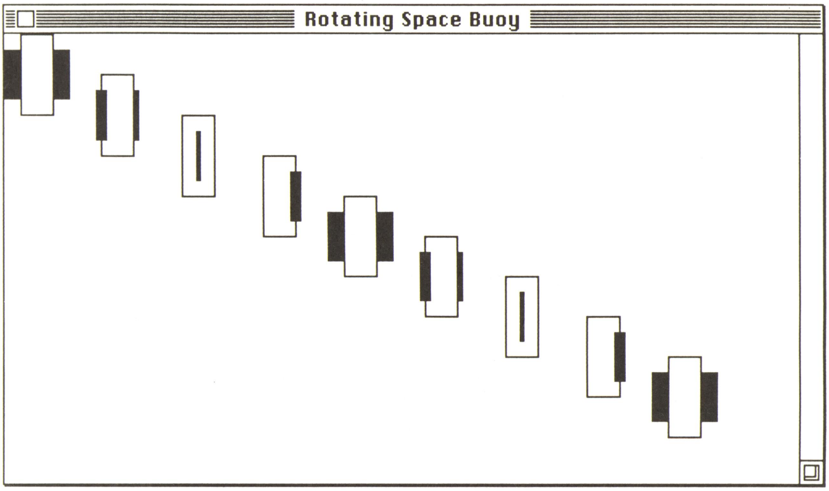
The idea of multiple images is to store a series of views of an object—each one slightly different—in an array, and then play them back rapidly so it looks like the image is moving. The method for accomplishing effects like this is more easily demonstrated than explained. So, type in and run the program shown in the listing in Figure 17-13.

FIGURE 17-13

The Rotating Space Buoy program



```
'________________________________________________ rotating space buoy

OPTION BASE 1
DIM view%(155,4)

CLS                                                      'first view
LINE (10,0) - (30,50), , b
LINE (0,10) - (10,40), , bf
LINE (30,10) - (40,40), , bf
GET (0,0) - (40,50), view%(1,1)

CLS                                                     'second view
LINE (10,0) - (30,50), , b
LINE (7,10) - (13,40), , bf
LINE (30,10) - (33,40), , bf
GET (0,0) - (40,50), view%(1,2)

CLS                                                      'third view
LINE (10,0) - (30,50), , b
LINE (19,10) - (21,40), , bf
GET (0,0) - (40,50), view%(1,3)

CLS                                                     'fourth view
LINE (10,0) - (30,50), , b
LINE (10,0) - (10,40), , bf
LINE (27,10) - (33,40), , bf
GET (0,0) - (40,50), view%(1,4)

CLS                                                'rotating display
FOR posit = 0 TO 200
   rot = (posit MOD 4) + 1               'find which view to show (1...4)
   PUT (posit * 2, posit), view%(1,rot)                      'put image
   FOR t = 1 TO 300 : NEXT t                                'delay loop
   PUT (posit * 2, posit), view%(1,rot)     'put again = erase (XOR mode)
NEXT posit
```



This program begins by generating four different views of an object we can think of as a space buoy. It's a cylinder with two vanes sticking out the sides (possibly they're solar panels). Once each view is generated, it's stored in the array *view%*.

Notice that this is a *two-dimensional* array. The first dimension is the size of the array needed to hold the object, which fits in a 40-by-50 rectangle, calculated with the program we created in Figure 17-10. The second dimension holds the number of different views we're going to display. In this program we have four views, but often there will be many more views of an object. The more views there are, the smoother the resulting motion of the object will be. Note that each GET statement addresses a different array element, from 1 to 4.

By using the OPTION BASE statement, we make the smallest index in our array equal to 1 rather than 0, so we can number our four views from 1 to 4, rather than from 0 to 3. This is sometimes useful to make it easier to keep track of what's going on.

As you can see (with a little imagination) from these views, the buoy is turning; the vane on the left is coming toward you, while the one on the right is receding and disappearing behind the buoy. The right vane in the third view has disappeared behind the buoy, and the left one is seen edge-on, sticking out toward us. In the fourth view the right vane has reappeared on the left, while the left vane is moving away from us on the right.

When we run the program, the FOR...NEXT loop with the *posit* counter variable displays these four images, one after the other. The variable *posit* is used to both determine the position of the buoy, and to find out which view of the buoy is next. The MOD operator in the second line of the loop takes *posit* and sets *rot* to a number between 1 and 4. The effect of this is that every fourth position of the buoy shows the same view. Next, *posit* is used in the PUT statement to set the starting point for the image, moving twice as far in the (x) direction as in the (y) direction. Notice that *rot* is used as an index in the PUT statement to select the four images, one after the other, from the *view%* array. The effect is that the space buoy appears to be rotating.

Actually, the computer is so fast that we need to slow the rotation to make it visible to us humans. That's the purpose of the program line:

```
FOR t = 1 TO 300 : NEXT t
```

This delay is arrived at by trial and error until the rotation finally looks as smooth as possible.

The delay makes possible the use of a somewhat different technique than we've used before to erase the old copy of the image. Before, we remembered the location of the old image (in a variable such as *oldPos*), and erased it before drawing the new one. In this case, because of the delay, we don't need to remember anything. We display the image, delay, and then erase it, using an exact copy of the statement we drew it with.

When we run the program, we first see the four views generated in the upper left corner of the Output window. Each one is erased in turn by a CLS statement, and finally the rotating buoy appears at the upper left corner, moving down toward the lower right corner.

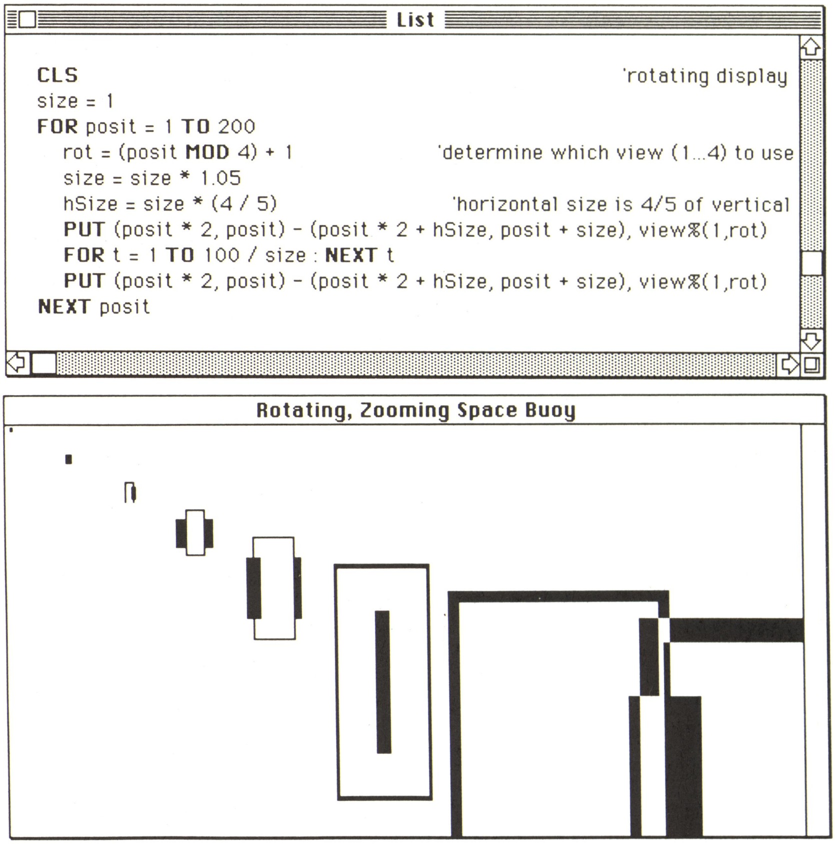
ROTATING AND ZOOMING

Let's put two of the techniques we've learned together: zooming and multiple views. We'll do this by making some small modifications to the last program, as shown in the listing in Figure 17-14 on page 450.

In this program we store the four images of the buoy in the same way as before. However, in the part of the program that displays the images, we introduce the variables *size* and *hSize* into the PUT statements. Without these variables, both the upper left and lower right corners of the image would be located in the same place relative to the upper right corner of the rectangle. In fact, *size* starts out as 1, so the resulting image is very small: only 1 pixel wide and 1 pixel high. However, *size* grows larger each time through the loop, with the result being that the lower right corner moves down and right. The effect is that the image "zooms" toward us as it rotates on its axis.

Notice that the *size* variable increases *nonlinearly.* That is, instead of adding a fixed amount to it each time through the loop, we *multiply* by a constant factor of 1.05 each time. This makes the image increase in size faster as it gets bigger. If the image increased at a constant rate, it would appear to increase its size rapidly when it was small, but as it grew larger it would appear to be growing much more slowly. As a general rule for this class of movement we might say that the rate of increase in size should be proportional to the size of the object.

Another wrinkle in this program is that we shorten the delay time as the image grows larger. The reason we need to do this points up a fundamental problem with the PUT statement: The larger the image PUT is



```
CLS                                                   'rotating display
size = 1
FOR posit = 1 TO 200
   rot = (posit MOD 4) + 1                 'determine which view (1...4) to use
   size = size * 1.05
   hSize = size * (4 / 5)                   'horizontal size is 4/5 of vertical
   PUT (posit * 2, posit) - (posit * 2 + hSize, posit + size), view%(1,rot)
   FOR t = 1 TO 100 / size : NEXT t
   PUT (posit * 2, posit) - (posit * 2 + hSize, posit + size), view%(1,rot)
NEXT posit
```

FIGURE 17-14

The Rotating, Zooming Space Buoy program

called upon to display, the longer it takes to display it. Thus the small images of the space buoy rotate rapidly, but as the buoy becomes larger, it rotates more and more slowly. To help compensate for this effect, we've changed the delay loop statement to read:

```
FOR t = 1 TO 100 / size : NEXT t
```

When *size* starts at 1, the delay will be 100 loops; but as *size* grows toward its final value, the delay will become smaller and smaller, until the cycle is only executed once between views. There's a moral here: Don't make your images too big, or it will take PUT a long time to display them, probably showing a lot of flicker, and your animation won't go as fast as you probably want it to.

Unfortunately, the buoy as it is shown in Figure 17-14 can't really give anything like an adequate impression of the resulting moving display. The display can only be appreciated on the Macintosh screen. Perhaps we can conclude from this that, if one picture is worth a thousand words, then one animation must be worth a thousand pictures.

INTERFACING TO MACPAINT

Images generated by computer programs, as the examples so far in this chapter have been, must of necessity be fairly simple. It's hard to imagine the BASIC statements that would generate a real-looking fish with detailed scales, or a Japanese lady with long flowing hair, or a detailed side view of a tennis shoe. But if we *could* use such images in our programs by storing them in arrays and using PUT statements to display them, it would increase our animation powers immensely. Imagine a pair of tennis shoes walking across the screen, or a Japanese lady combing her hair.

Happily, the means to create images of great complexity and realism already exist for the Macintosh, in the MacPaint program. The question is, how do you transfer an image that you have created with MacPaint into a BASIC program?

The answer to this question involves a number of steps. We'll list them now, then go back and explain each step in detail.

1. Draw the picture in MacPaint.
2. Copy the picture to the Clipboard.
3. Execute a BASIC program to read the image from the Clipboard into a disk file.
4. Execute a final BASIC program to read the image from the file onto the screen, where it can be stored in an array using GET.

Drawing the image and copying it to the Clipboard

If you're familiar with MacPaint, you won't have any trouble with this process. Keep in mind that PUT works faster with small images; as the images get larger, it becomes harder to move them rapidly. So it's wise not to start with a large image.

To copy something from MacPaint to the Clipboard, we first surround it with the Marquee (the box outlined by the blinking dashed line). Then we choose Copy from the Edit menu, which puts the selected image on the Clipboard. Finally, we quit MacPaint and return to the Mac desktop. After we're back on the Macintosh desktop, we need to copy the Clipboard file from the MacPaint disk to our BASIC work disk by dragging its icon (usually stored in the System folder) to the BASIC disk. From this point on, we can't Cut or Copy anything else to the Clipboard, until we've stored the image on it in a file created by our BASIC program.

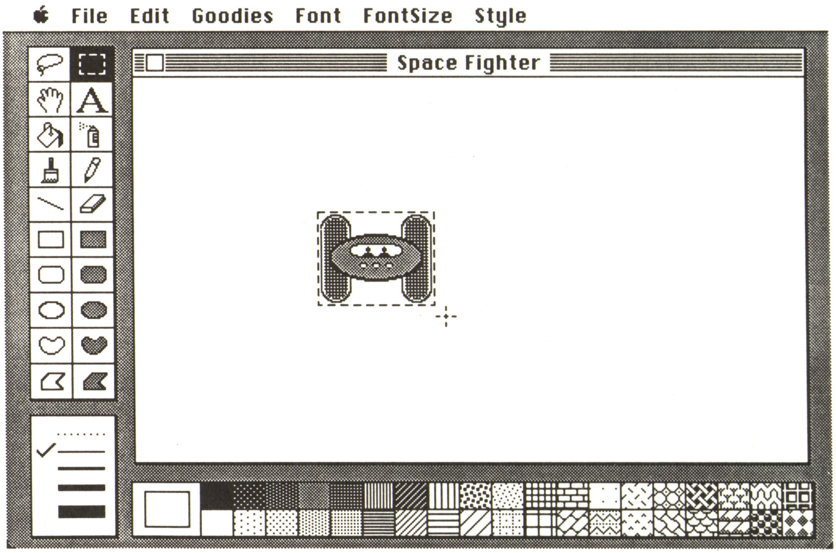


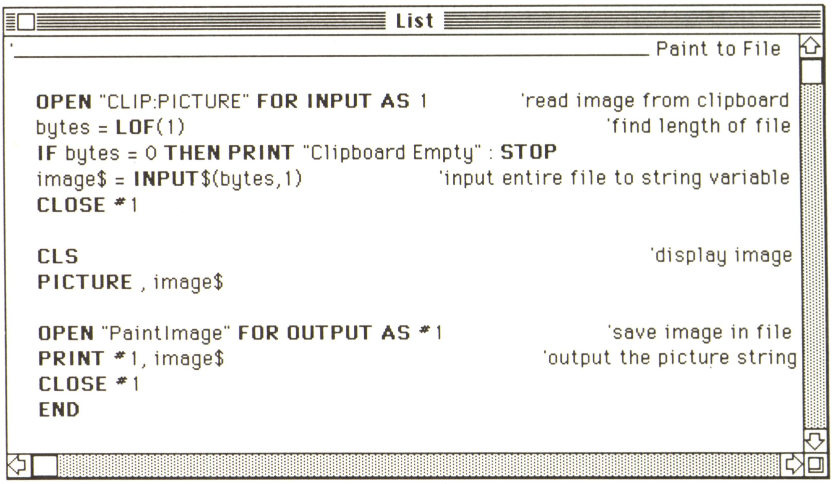
FIGURE 17-15

An image created in MacPaint

An example of an image created in MacPaint is shown in Figure 17-15, which represents an "intergalactic space fighter," seen head-on. This is not an elaborate design; it is included here only to give you the idea of how the transfer process works. We leave it to you to draw your own version of a full-scale starship, or perhaps the inside of a bacterium, or a steam engine.

Reading the image from the Clipboard to a disk file

To read the image from the Clipboard to a disk file we must execute a series of BASIC statements. These statements are contained in the program called Paint-to-File, shown in Figure 17-16.



```
'________________________________________________ Paint to File

 OPEN "CLIP:PICTURE" FOR INPUT AS 1            'read image from clipboard
 bytes = LOF(1)                                    'find length of file
 IF bytes = 0 THEN PRINT "Clipboard Empty" : STOP
 image$ = INPUT$(bytes,1)               'input entire file to string variable
 CLOSE #1

 CLS                                                    'display image
 PICTURE , image$

 OPEN "PaintImage" FOR OUTPUT AS #1                     'save image in file
 PRINT #1, image$                                'output the picture string
 CLOSE #1
 END
```

FIGURE 17-16

A listing of the Paint-to-File program

Although short, this program uses several new ideas. We'll assume you've read Chapter 16, "Files," and understand the use of such statements as OPEN and CLOSE.

Reading the image from the Clipboard

The first half of the program reads the image from the Clipboard and stores it in a string variable called *image$*. This is something new: We've seen that an image can be stored in an array; now we see that it can be stored in a string as well. Why not store it in an array again? Because that's not how it's stored on the Clipboard. The Clipboard actually records the graphics commands used to create the image, and not the image itself. So, a string variable is the right form for storing these commands.

BASIC can read from the Clipboard just like any other disk file, using the statement:

```
OPEN "CLIP:PICTURE" FOR INPUT AS 1
```

CLIP means Clipboard. PICTURE, used in this statement, means that the data on the Clipboard are to be interpreted as a series of graphics statements. If the word TEXT had been used instead, it would mean we were expecting to transfer ordinary text from the Clipboard, copied there from a word-processing program such as MacWrite or Microsoft Word. Because the contents of the Clipboard can be in more than one format, we need to have our program verify that it's in the format we desire. Since we've opened the file as a picture, the LOF(1) function (short for Length Of File) returns the length of the file in bytes and assigns it to the variable *bytes.* But if the Clipboard is empty, or it contains text, the LOF function returns the value zero and the IF...THEN statement stops the program.

Once the Clipboard has been opened as a file, its contents are read in two steps. Our program uses LOF to find out how many bytes of information are to be read, and stores the resulting number in the variable *bytes.* Then, using this variable, it reads all the Clipboard data with the INPUT$ statement.

Reading the image: The INPUT$ function

Recall that in Chapter 16 we used the INPUT statement to read data from a file one *field* at a time. However, data are not always stored in a file as delimited or fielded records. When we need to get data from a file that are not necessarily stored as records, we must use the INPUT$ function, which reads a file a specified number of *bytes* at a time. The syntax of the INPUT$ function is:

string variable name = **INPUT$**(*n, File Number*)

where *n* is the specified number of *bytes.*

When you cut or copy a MacPaint image to the Clipboard, it's stored as a stream of bytes in PICTURE format. So in our current program, we use INPUT$ to read the entire contents of the Clipboard file into the variable *image$.* Finally, we close the file.

Displaying the image: The PICTURE statement

In the past we've used PUT to transform an image stored in an array into a visible picture on the screen. How do we display an image stored in a string? The answer is the PICTURE statement. This statement interprets the graphics statements contained in the string *image$,* and translates them into a picture on the screen.

The PICTURE statement actually has some capabilities we don't need to use in our present program. For instance, we don't need to place the image in the upper left corner of the screen if we don't want to. By adding parameters to PICTURE, we can put the image anywhere on the screen. Also, if we want, we can use PICTURE to change the size and shape of the image, in much the same way as we can with PUT. The format for doing these things can be found in the BASIC manual; in our present program the exact image in the upper left corner is all we need.

Also note that the graphics image to be interpreted by PICTURE doesn't need to come from the Clipboard. We learned in Chapter 14 that it is also possible to record a series of graphics commands created by a BASIC program. This is done by executing the statement PICTURE ON, then issuing some graphics statements such as LINE and CIRCLE, and then saying PICTURE OFF. The resulting image can be captured in a string variable, and can be displayed with PICTURE. However, this process is not relevant for us at this point.

Saving the image in a disk file

Now that the graphics commands that constitute our image are in a string, what are we going to do with them? We want to put them somewhere so that another BASIC program, the one that's actually going to perform the animation, can get at them. Clearly we can't rely on the image always being on the Clipboard. The answer is to store the graphics commands as a string in its own file. Then any BASIC program can access that file when it wants to "see" the image.

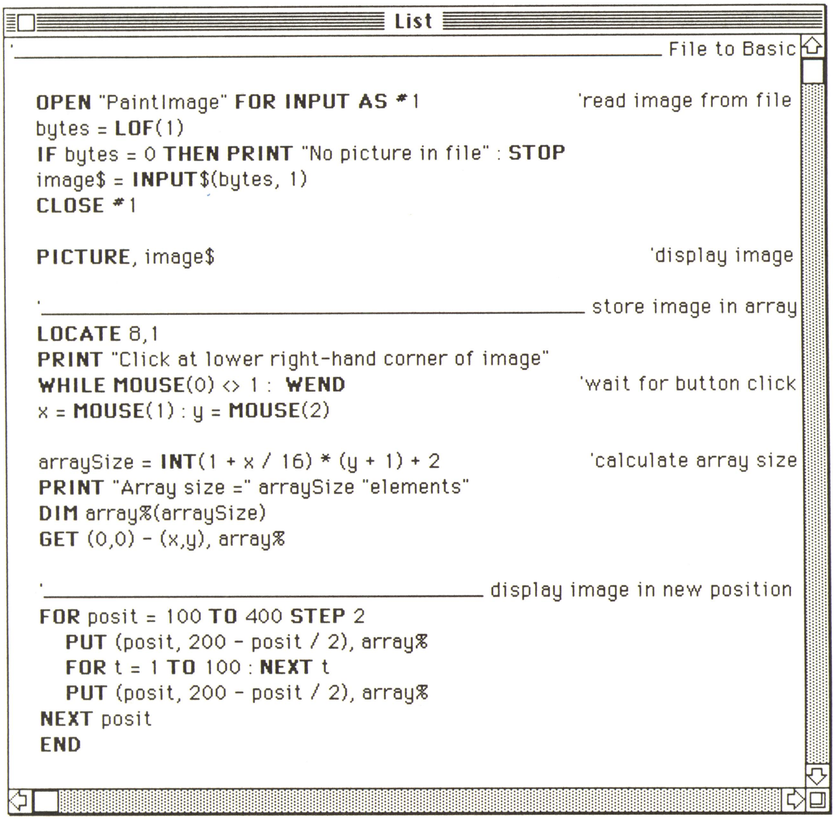
To save these graphics commands, we open a sequential file called *PaintImage,* and use a PRINT# statement to write the string *image$* to it.

In our particular case, having copied an image from MacPaint to the Clipboard and copied the Clipboard file to the BASIC disk, we simply run the Paint-to-File program to store the image as a file.

Reading the image from a disk file to an array

Now that the image is safely stored as a set of graphics commands in a file, our program has only to read the image back in from the file and display it on the screen. Once there, another program can operate on it with GET and PUT to store the image in an array and put it back on the screen. The listing shown in Figure 17-17 is a program that does just this.

Our program first reads the image from the file into the variable *image$,* in much the same way as the last program read it from the Clipboard into a variable with the same name. It then displays the image. This



```
'_____________________________________________________________ File to Basic

OPEN "PaintImage" FOR INPUT AS #1                        'read image from file
bytes = LOF(1)
IF bytes = 0 THEN PRINT "No picture in file" : STOP
image$ = INPUT$(bytes, 1)
CLOSE #1

PICTURE, image$                                                  'display image

'_____________________________________________________ store image in array
LOCATE 8,1
PRINT "Click at lower right-hand corner of image"
WHILE MOUSE(0) <> 1 :  WEND                             'wait for button click
x = MOUSE(1) : y = MOUSE(2)

arraySize = INT(1 + x / 16) * (y + 1) + 2                'calculate array size
PRINT "Array size =" arraySize "elements"
DIM array%(arraySize)
GET (0,0) - (x,y), array%

'_________________________________________ display image in new position
FOR posit = 100 TO 400 STEP 2
   PUT (posit, 200 - posit / 2), array%
   FOR t = 1 TO 100 : NEXT t
   PUT (posit, 200 - posit / 2), array%
NEXT posit
END
```

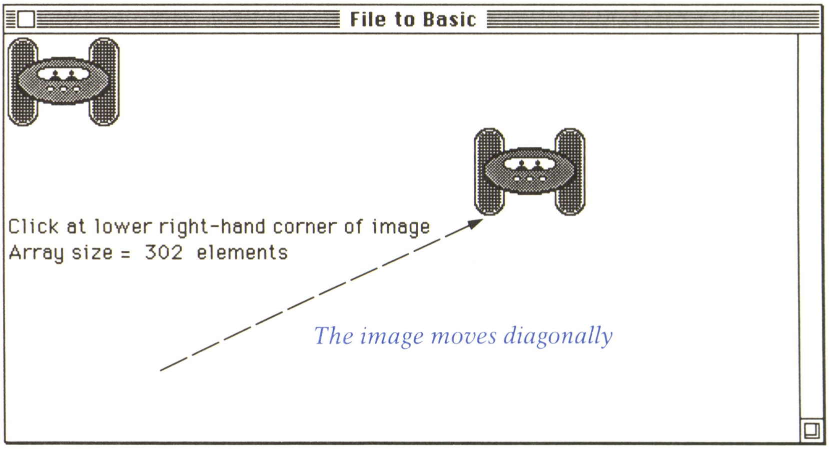
FIGURE 17-17

A listing of the File-to-BASIC program is necessary because, as in the programs earlier in the chapter, we'll want to use GET to read the image from the screen and store it in an array. So we use the PICTURE statement, as explained earlier, to put the image on the screen.

From this point on, our program uses techniques we've seen in action earlier in the chapter. We wait in a WHILE...WEND loop for the mouse to be clicked in the lower left corner of the object so we can determine the size of the array needed to hold it. (This part of File-to-BASIC would not become a permanent part of our program; we would only execute these instructions once, to see how large a number to put in the DIM statement.) Figure 17-18 shows how the screen looks when we find the image size with the mouse.

Finally, we use GET to store the image in the array, and PUT to read it back onto the screen and move it across the output window.

In practice, we would run File-to-BASIC once, to find out the array size. Then we would take the lines at the beginning of the program, which read the image from the file into the string variable *image$,* the PICTURE statement to put the image on the screen, and a GET statement to store the image in an array, and incorporate them into our program. Then our program would use PUT statements, as we've shown in other programs in this chapter, to perform whatever animation effects we wanted on the resulting MacPaint image.



```
Click at lower right-hand corner of image
Array size =  302  elements
```

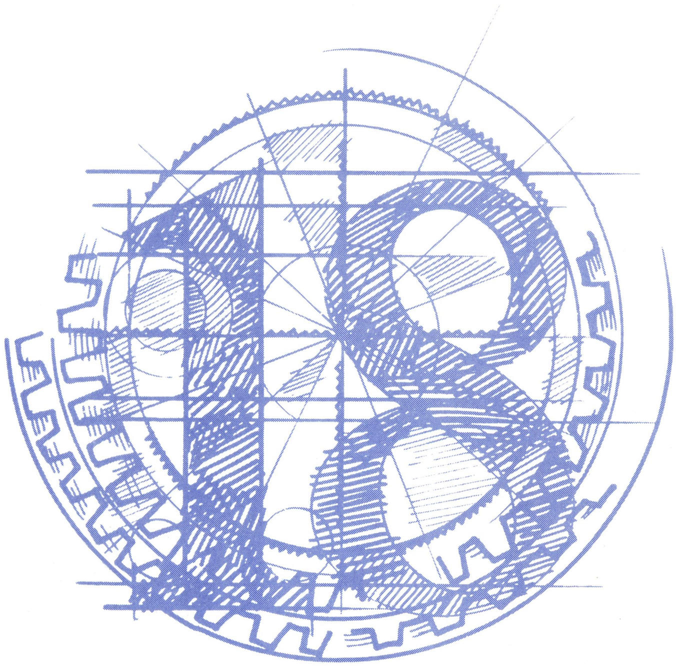
FIGURE 17-18

Finding the size of the image in the File-to-BASIC program

REFRESH-YOURSELF WINDOW

In this chapter you've learned the rudiments of animation using Microsoft BASIC. You've learned how to store images in arrays so they can be accessed by the PUT statement, and how to use PUT to move the image on the screen, and change its size and shape. You've also learned how to store multiple images of an object using a two-dimensional array and an index in the PUT statement, so that objects may be made to appear to be rotating and moving in other complex ways. Finally you've learned how images created with MacPaint may be incorporated into your program.

This chapter has only scratched the surface of animation in Microsoft BASIC for the Macintosh. But you should be able to build on what you've learned here and go on to create whatever effects you want.



Multivoice Sound

Although we've seen a great deal of what Microsoft BASIC can make the Macintosh do, we haven't heard *much from the Mac outside of a few beeps. But with BASIC, we can add sound to our programs to make them more interesting and entertaining. We used the SOUND command briefly in Chapter 2; this chapter goes into much more depth about how to add sound* effects and multivoice music to our Microsoft BASIC programs. It first discusses how to make simple sounds with the SOUND command. We will then see how to combine SOUND commands to mix more than one sound to produce *multivoice* sounds. Multivoice sounds are almost as easy to program as single-voiced sounds. Finally, we will learn how to change the "texture" of the sounds that we produce to make sounds with distinct identities, as well as many examples that demonstrate the effective use of sound in programs.

Before we begin, we should mention that a few Macintoshes have flaws that cause poor sound reproduction. If you don't hear anything when you run one of the programs in this chapter, or the sound is very soft or very fuzzy, your Macintosh may need servicing. And even if you don't have one of these problems, you will get much better sound reproduction if you attach an external speaker or stereo system to the jack in the back of your Macintosh that has a musical-note icon above it.

HOW SOUND IS GENERATED

Unlike the many graphics commands that we use to put images on our screen, we only need one Microsoft BASIC command to actually generate sound: the SOUND command. The other command that we can use, the WAVE command, sets up the sound-generation hardware in the Macintosh for multivoice sounds and to control the texture of the sound.

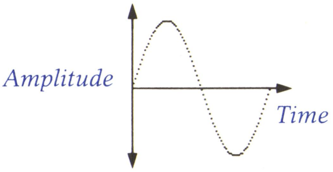
The general syntax of the SOUND command is:

SOUND *Frequency, Duration, Volume, Voice Number*

The first two parameters, *Frequency* and *Duration,* are required each time we give the SOUND command, while *Volume* and *Voice Number* are optional. All of the parameters are fairly easy to understand even if you are not familiar with how sounds are produced.

The Frequency parameter: Specifying the pitch of the sound

When we hear music, the feature that makes two notes different is their *frequency.* In scientific terms, the frequency is a measurement of the number of cycles per second. A cycle can be illustrated as:



The cycles per second are measured in *hertz,* or *Hz* for short. The range of normal human hearing is about 10 to 15,000 Hz, and most musical sounds are in the range of 50 to 5000 Hz. The *Frequency* parameter of the SOUND command simply controls the number of cycles per second for any sound we want to make. As you might expect, the value of this parameter must be positive.

The Duration parameter: Specifying how long the sound will play

After we tell Microsoft BASIC the frequency of the sound we want it to make, we also have to tell it how long the sound should last. The *Duration* parameter has somewhat strange units: a value of 18.2 will play the sound for one second. We can specify a *Duration* value from 0 to 77, which is a range of 0 to 4.2 seconds. A short audible duration, 1, is approximately 55 milliseconds long. (These values are similar to Microsoft BASIC on the IBM PC.)

The Volume parameter: Specifying the loudness of the sound

If we are producing a single sound, we can change the volume that Microsoft BASIC uses for the sound. The *Volume* parameter can range from 0 (silence) to 255 (very loud); if we do not specify a *Volume,* Microsoft BASIC will play the sound as if we had specified 127. The *Volume* option is modified by the volume setting on the Macintosh Control Panel, so that if we have the Control Panel set high and the *Volume* option set above 127, the sound produced will be very loud.

The Voice Number parameter: Specifying which voice to use

As mentioned earlier, we can write programs that generate multivoiced sounds. A typical example of using multivoice sound is playing musical *chords,* which are collections of musical notes that sound pleasurable to the ear when played simultaneously. The Macintosh has four voices numbered 0 through 3. We cannot use any voice other than 0 until we have given a WAVE command to define one of the other voices; this is explained in detail later in the chapter.

EXAMPLES OF USING SOUNDS IN PROGRAMS

When we program with simple sounds, all we usually need to specify are the *Frequency* and *Duration* parameters, which are the fundamental elements of sound. For example, to sound a tone of 500 Hz for 1 second, we would give the command:

```
SOUND 500, 18.2
```

It's easy to forget the ratio for converting the value of the *Duration* parameter to the actual length of time a sound is produced, so you may want to set up a variable that contains the conversions. For instance, the following program plays a 10,000 Hz "squeak" for ⅓ second:

```
durConv = 18.2
SOUND 10000, .33 * durConv
```

Adjusting the volume of the sound is also straightforward. Recalling that *Volume* is the third SOUND parameter, we can make very loud, obnoxious sounds with:

```
SOUND 50, 25, 255
```

or quiet, subtle sounds with:

```
SOUND 5000, 25, 50
```

MAKING SINGLE SOUNDS

The two most common uses for sounds in BASIC programs are for sound effects and music. Sound effects can be anything that we can imagine, such as alerts, laser blasts, screeching tires, rocket motors, ticking clocks, and so on. There are many uses for music in programs, such as a short melody when the user accomplishes something or a beautiful Bach Prelude while our program's title screen is shown. We can create a wide variety of sound effects and music with just single-voice sounds, as we'll discover in the following sections.

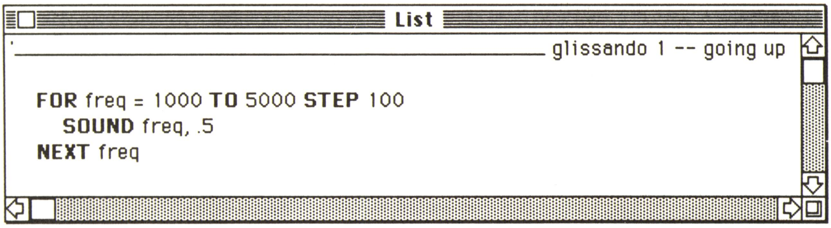
Creating single-voice sound effects

The sound effects we hear in a video arcade and in modern science-fiction films are usually made from complex multivoice generators. However, until a few years ago sound effects were often made with only one voice. The game that started the video revolution, Pong, had only one voice to make the "beep" and "boop" sounds that you heard when you played.

A single voice can create a wide variety of sounds if its tone changes quickly enough, which, fortunately, the Macintosh is quite capable of doing. Thus, we can make a number of interesting sound effects using just a single voice.

FIGURE 18-1

A simple glissando program



```
'______________________________________ glissando 1 -- going up

FOR freq = 1000 TO 5000 STEP 100
  SOUND freq, .5
NEXT freq
```

A common sound effect that is easy to program is a *glissando.* A glissando is a sound characterized by a rapidly increasing or decreasing frequency. For instance, a slide whistle creates a glissando when we push or pull the slide as we blow.

We can create a glissando with a FOR...NEXT loop that steps through the frequencies that we want to hear. Figure 18-1 shows a simple glissando program.

Notice that the frequency is set by the *freq* variable that increases in the FOR...NEXT loop. However, glissandos can go down in frequency as well as up. To do this in our program, we need only reverse the initial and final indexes of the FOR...NEXT loop like this:

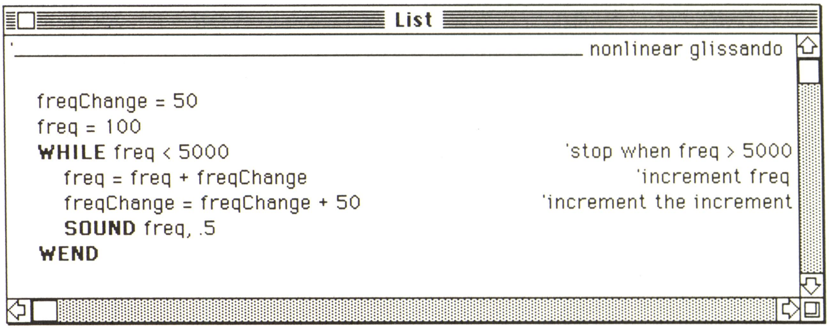
```
FOR freq = 5000 TO 1000 STEP -100
```

To make a glissando glide faster, we can simply alter the rate of the frequency change. In the preceding two examples, each step of the loop changed the frequency by 100 Hz. To create a faster sound effect, we can simply change the FOR...NEXT loop like this:

```
FOR freq = 5000 TO 1000 STEP -300
```

This increases the step size of the loop so that it reaches its end faster.

We can make the glissando even more dramatic by *accelerating* the "rate" at which we step through the frequencies. Figure 18-2 on page 464 shows a program that does this. We used a WHILE...WEND loop and created a variable that controls how fast the *freq* variable increments. The result sounds like a space weapon or something falling.



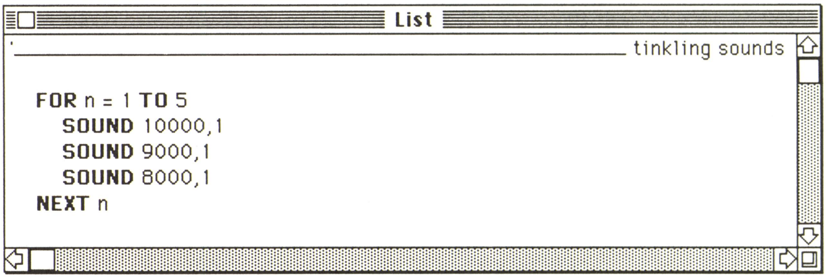
```
'______________________________________________ nonlinear glissando

freqChange = 50
freq = 100
WHILE freq < 5000                                  'stop when freq > 5000
   freq = freq + freqChange                              'increment freq
   freqChange = freqChange + 50                'increment the increment
   SOUND freq, .5
WEND
```

FIGURE 18-2

A glissando program with accelerating change of frequency

Of course, there are many fun sound effects other than glissandos. For example, we may want to add a tinkling noise to a program that has an object flitting across the screen. Figure 18-3 shows an easy way to make a high-pitched tinkling sound.

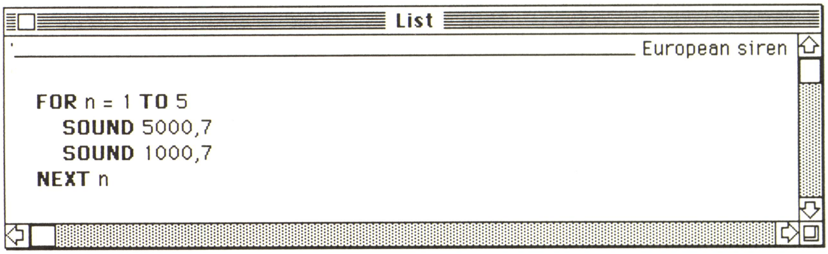
It's often fun to experiment with the sound effects that we create. For example, we can slightly modify the tinkling sound to make it sound more like a European warning siren, as in Figure 18-4. Here, we alternate between two very different frequencies, 1000 and 5000 Hz, playing each for about ½ second.



```
'__________________________________________________ tinkling sounds

FOR n = 1 TO 5
   SOUND 10000,1
   SOUND 9000,1
   SOUND 8000,1
NEXT n
```

FIGURE 18-3

A program that creates tinkling sounds



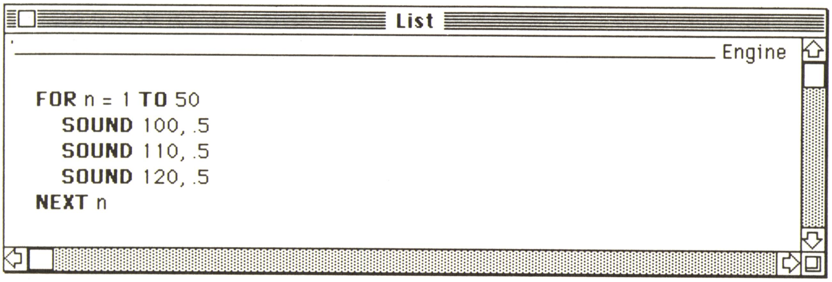
```
'__________________________________________________ European siren

FOR n = 1 TO 5
   SOUND 5000,7
   SOUND 1000,7
NEXT n
```

FIGURE 18-4

A European-siren program

FIGURE 18-5

An engine-sound program



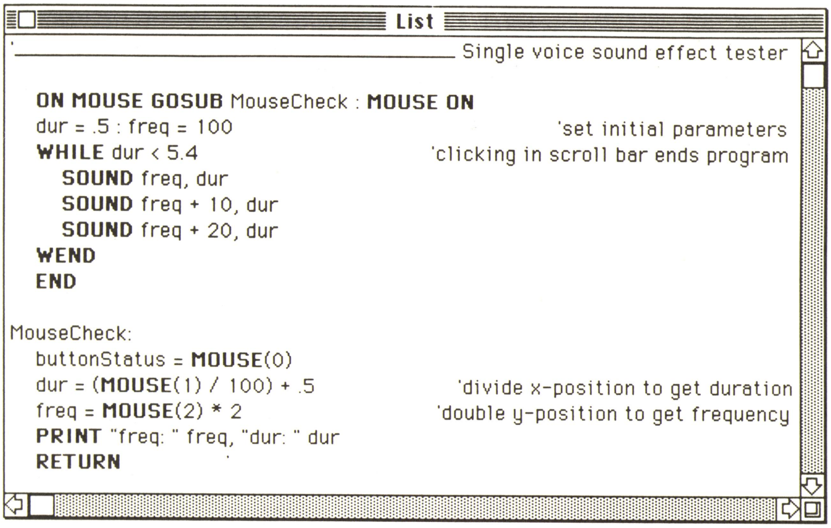
```
'                                                  Engine

  FOR n = 1 TO 50
    SOUND 100, .5
    SOUND 110, .5
    SOUND 120, .5
  NEXT n
```

We can toy with several low frequencies to create other interesting single-voice sound effects. In Figure 18-5, we use frequencies around 100 Hz to make a sound like a motor or engine running.

Experimenting with several different parameters of the SOUND statement simultaneously often turns up weird and amusing textures and “soundscapes.” Figure 18-6 is a simple program that allows us to play with repetitive sounds that are clustered around a single frequency. The program takes off on the engine effect, except that the base frequency for the three SOUND statements and the durations are controlled by the position of the pointer when the mouse button is pressed. Moving the mouse vertically on the screen and clicking the button affects the frequency, while moving it horizontally affects the duration. The event-trapping feature described in Chapter 11 is used so that the program reacts instantly to the

FIGURE 18-6

A program that uses the mouse to create sounds of different duration and frequency

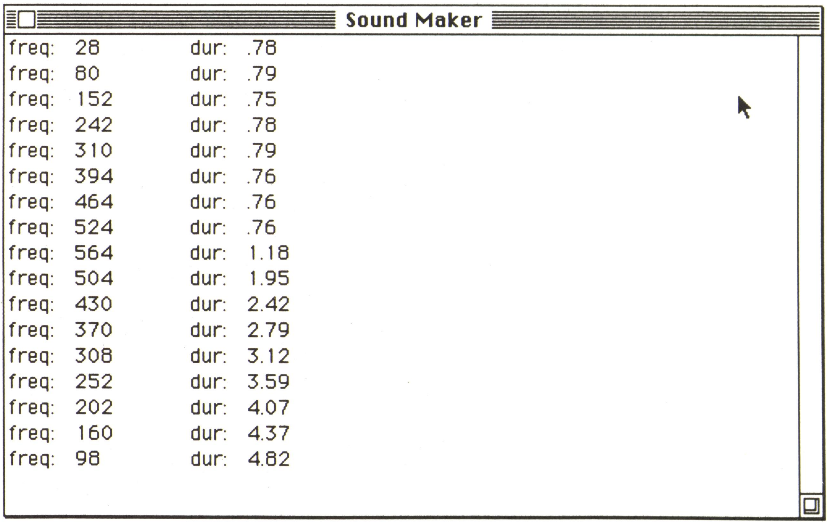


```
'                                     Single voice sound effect tester

  ON MOUSE GOSUB MouseCheck : MOUSE ON
  dur = .5 : freq = 100                         'set initial parameters
  WHILE dur < 5.4                          'clicking in scroll bar ends program
    SOUND freq, dur
    SOUND freq + 10, dur
    SOUND freq + 20, dur
  WEND
  END

MouseCheck:
  buttonStatus = MOUSE(0)
  dur = (MOUSE(1) / 100) + .5               'divide x-position to get duration
  freq = MOUSE(2) * 2                      'double y-position to get frequency
  PRINT "freq: " freq, "dur: " dur
  RETURN
```

(continued)

FIGURE 18-6

(continued)



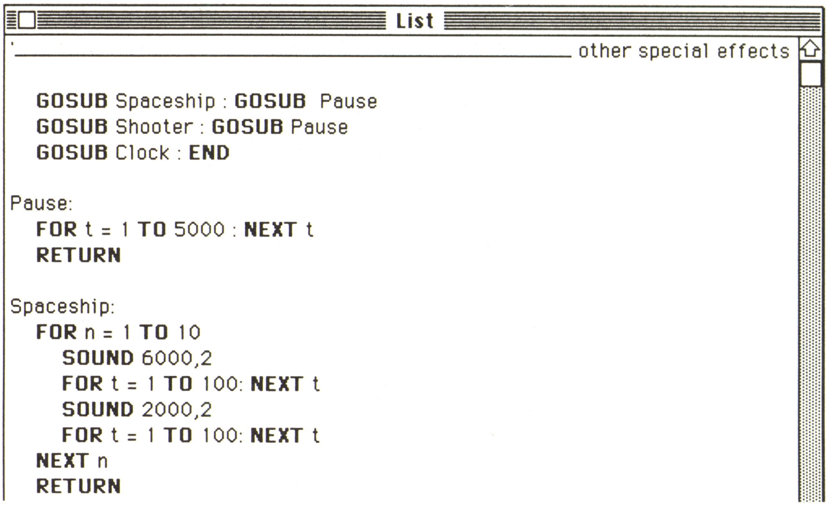
```
freq:  28          dur:  .78
freq:  80          dur:  .79
freq:  152         dur:  .75
freq:  242         dur:  .78
freq:  310         dur:  .79
freq:  394         dur:  .76
freq:  464         dur:  .76
freq:  524         dur:  .76
freq:  564         dur:  1.18
freq:  504         dur:  1.95
freq:  430         dur:  2.42
freq:  370         dur:  2.79
freq:  308         dur:  3.12
freq:  252         dur:  3.59
freq:  202         dur:  4.07
freq:  160         dur:  4.37
freq:  98          dur:  4.82
```

mouse button and the loop is not delayed by continual polling of the MOUSE function. The values for *dur* and *freq* are displayed in the Output window so we can incorporate them into another program.

As you experiment with the SOUND command, you may want to save your sounds in a collection of subroutines or subprograms so you can use them in different programs. The sound effects shown in Figure 18-7 should help you start an interesting collection. You can later try different variations on the sounds that you create, such as changing the frequency or making an effect run slower.

FIGURE 18-7

A variety of sound-effect subroutines

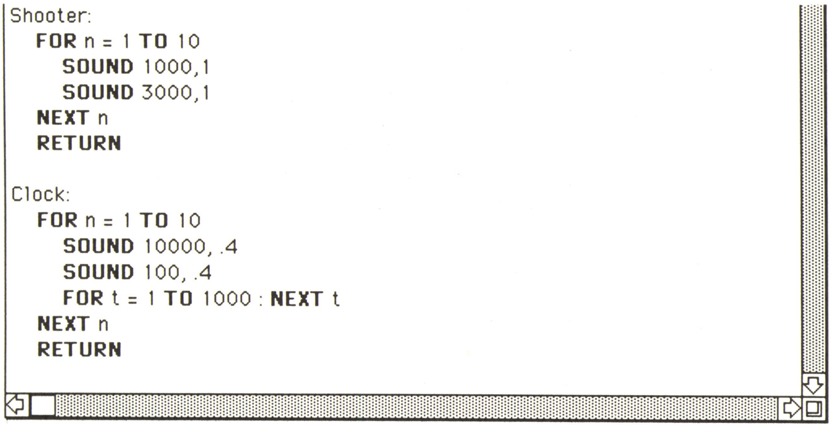


```
'___________________________________________ other special effects

  GOSUB Spaceship : GOSUB  Pause
  GOSUB Shooter : GOSUB Pause
  GOSUB Clock : END

Pause:
  FOR t = 1 TO 5000 : NEXT t
  RETURN

Spaceship:
  FOR n = 1 TO 10
    SOUND 6000,2
    FOR t = 1 TO 100: NEXT t
    SOUND 2000,2
    FOR t = 1 TO 100: NEXT t
  NEXT n
  RETURN
```

(continued)

FIGURE 18-7

(continued)



```
Shooter:
  FOR n = 1 TO 10
    SOUND 1000,1
    SOUND 3000,1
  NEXT n
  RETURN

Clock:
  FOR n = 1 TO 10
    SOUND 10000, .4
    SOUND 100, .4
    FOR t = 1 TO 1000 : NEXT t
  NEXT n
  RETURN
```

Playing musical notes

If the sounds of zapping aliens or racing Corvettes don't appeal to you, you may be more interested in using Microsoft BASIC to make music. Since the musical scale is simply a special set of frequencies, we can play music with the SOUND command by choosing the correct frequencies.

Even if you are unfamiliar with music theory, you can use your Macintosh to play wonderful music. In fact, you may become inspired enough to learn more about music so that you can play it with Microsoft BASIC. If you don't know (or don't remember) much about music theory, here is a very brief description to get you through this chapter:

- A musical note is a single frequency. Only certain frequencies are considered musical notes.
- Each note has a letter, A through G. An A note has a lower frequency than a B note, which is lower than a C note, and so on.
- Three or more notes can be played together to form a *chord*. Some chords are more pleasing to the ear than others.
- Some notes have either a *sharp* (indicated by a # symbol) or a *flat* (indicated by a ♭) attached to their letter. A sharp means that the note is a little higher in frequency than the named note, and a flat means that the note is a little lower in frequency. For example, A# is the note between A and B, and is the same as B♭ (B and E do not have sharps.)
- The set of the 12 notes between C and B is called an *octave*. The next set of frequencies after one octave is simply the next octave with the same lettering for the notes.

A complete octave consists of the following notes:

C C# D D# E F F# G G# A A# B

FIGURE 18-8

One octave of the equal-tempered scale on a piano keyboard

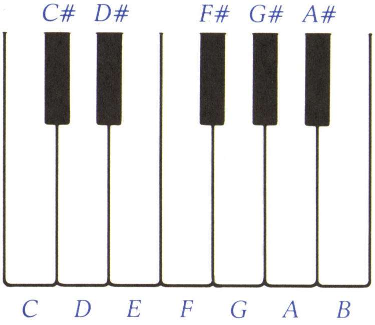


Figure 18-8 shows an octave on a piano keyboard. The actual sequence of frequencies for the scale we'll be working with is referred to as the *equal-tempered scale.* Other scales do exist, but this one is the most common in Western music.

The table in Figure 18-9 shows the frequencies for the complete equal-tempered scale. The table uses the standard used by most classical musicians, which has the frequency of the middle A note set at 440 Hz. For example, to play a C note as a quarter-note, we might use the table and these commands:

```
wholeNote = 18.2
SOUND 523.2, .25 * wholeNote
```

We can write whole sections of music using just SOUND commands with the respective notes.

To write a program that plays a song, we must first establish the notes of the song. For instance, the notes for "Yankee Doodle" are:

C C D E C E D G C C D E C B

Although it can't be indicated on this line, the G note is in a lower octave than the rest of the notes, and the last C note has a duration twice as long as the other notes. Figure 18-10 on page 470 shows a program that plays "Yankee Doodle."

If you refer to the table of frequencies in Figure 18-9 and study it for a few minutes, you'll find an interesting pattern emerges: The frequency for a note in one octave is exactly half the same note in the octave above and exactly twice the same note in the octave below. Thus, the frequency of the A note in the octave below the middle A is 220 Hz. This

FIGURE 18-9

A table of the musical frequencies in the equal-tempered scale

Note	Frequency	Note	Frequency	Note	Frequency
C0	16.35	G#2	103.83	E5	659.26
C#0	17.32	A2	110.00	F5	698.46
D0	18.35	A#2	116.54	F#5	739.99
D#0	19.45	B2	123.47	G5	783.99
E0	20.60	C3	130.81	G#5	830.61
F0	21.83	C#3	138.59	A5	880.00
F#0	23.12	D3	146.83	A#5	932.33
G0	24.50	D#3	155.56	B5	987.77
G#0	25.96	E3	164.81	C6	1046.50
A0	27.50	F3	174.61	C#6	1108.73
A#0	29.14	F#3	185.00	D6	1174.66
B0	30.87	G3	196.00	D#6	1244.51
C1	32.70	G#3	207.65	E6	1328.51
C#1	34.65	A3	220.00	F6	1396.91
D1	36.71	A#3	233.08	F#6	1479.98
D#1	38.89	B3	246.94	G6	1567.98
E1	41.20	C4	261.63	G#6	1661.22
F1	43.65	C#4	277.18	A6	1760.00
F#1	46.25	D4	293.66	A#6	1864.66
G1	49.00	D#4	311.13	B6	1975.53
G#1	51.91	E4	329.63	C7	2093.00
A1	55.00	F4	349.23	C#7	2217.46
A#1	58.27	F#4	369.99	D7	2349.32
B1	61.74	G4	392.00	D#7	2489.02
C2	65.41	G#4	415.30	E7	2637.02
C#2	69.30	A4	440.00	F7	2793.83
D2	73.42	A#4	466.16	F#7	2959.96
D#2	77.78	B4	493.88	G7	3135.96
E2	82.41	C5	523.25	G#7	3322.44
F2	87.31	C#5	554.37	A7	3520.00
F#2	92.50	D5	587.33	A#7	3729.31
G2	98.00	D#5	622.25	B7	3951.07
				C8	4186.01

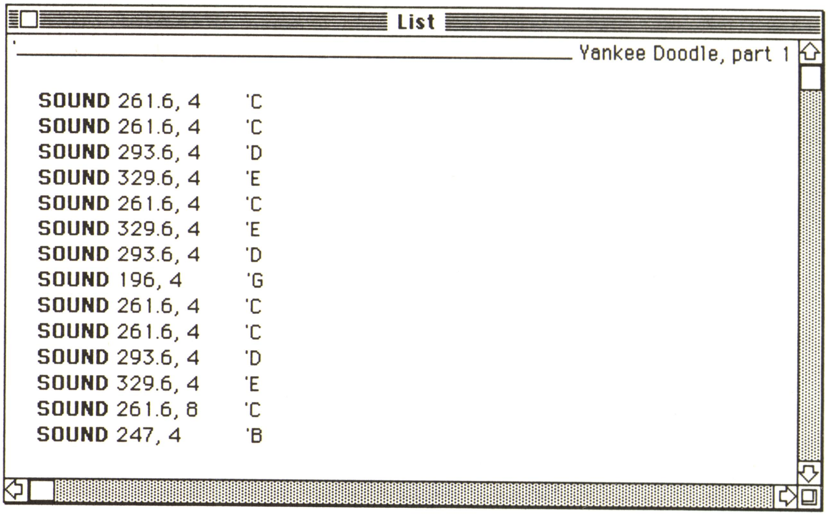
Note: Equal-Tempered Chromatic Scale; A4 = 440
American Standard pitch—adopted by the American Standards Association in 1936

means that we can raise or lower a sound an octave simply by multiplying or dividing its frequency by 2. Here are the first few lines of the previous "Yankee Doodle" program altered to raise the notes one octave.

```
SOUND 523.2, 4      'C
SOUND 523.2, 4      'C
SOUND 587.2, 4      'D
SOUND 659.2, 4      'E
```

FIGURE 18-10

A program that plays "Yankee Doodle"



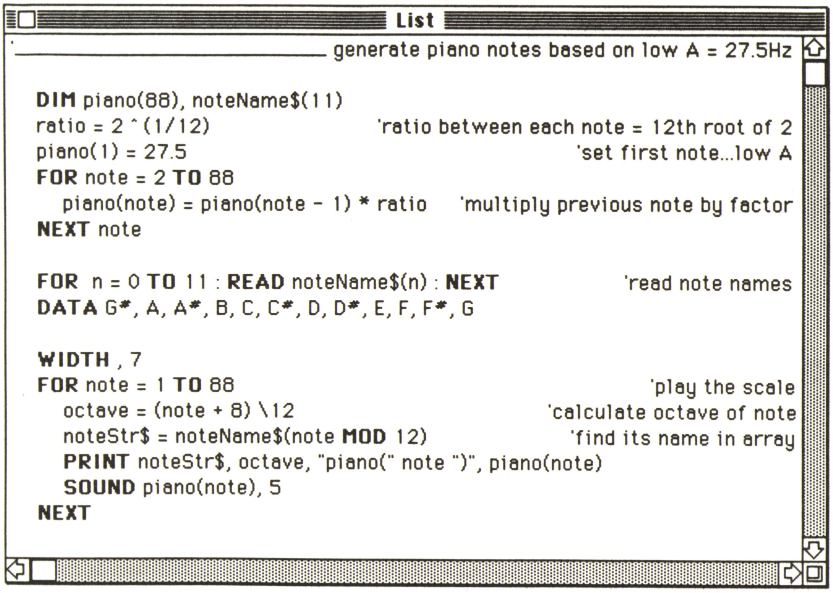
```
'————————————————————— Yankee Doodle, part 1

SOUND 261.6, 4     'C
SOUND 261.6, 4     'C
SOUND 293.6, 4     'D
SOUND 329.6, 4     'E
SOUND 261.6, 4     'C
SOUND 329.6, 4     'E
SOUND 293.6, 4     'D
SOUND 196, 4       'G
SOUND 261.6, 4     'C
SOUND 261.6, 4     'C
SOUND 293.6, 4     'D
SOUND 329.6, 4     'E
SOUND 261.6, 8     'C
SOUND 247, 4       'B
```

Since the starting point for each octave is twice as high as the previous octave, musical notes are said to be based on a *geometric scale* instead of a *linear scale.* (In a linear scale, each note would be related to the ones around it by a constant number of cycles per second.) Since each octave has twice the frequency of the previous one, the scale is derived from powers of 2. Each note differs from the next by a ratio of the 12th root of 2—that is, 1.05946.

(If you are unfamiliar with exponents or strange roots, you can skim the following section without worrying about the math. Type the programs in anyway, since the results of the programs will be interesting, even if the math isn't.)

It is easy to access the frequencies for notes in an array. A natural choice would be to have each element of the array be a note, so that each group of twelve elements in the array would be an octave. Figure 18-11 shows a routine that fills the array called *piano* with calculations of the 88 frequencies for the notes of the 88 piano keys.

Since the array is based on a value of an A note, *piano(1)* is a low A at 27.5 Hz, *piano(2)* is a low A# at 29.135 Hz, and so on. The table in Figure 18-12 shows the relationship between each note and the array. To transpose a note to a higher octave, simply add a multiple of 12 to the array index. For instance, the frequency of a C note in the first octave would be accessed as *piano(4),* while a C note in the second octave would be accessed as *piano(12+4),* or *piano(16).*



```
'____________________________ generate piano notes based on low A = 27.5Hz

DIM piano(88), noteName$(11)
ratio = 2 ^ (1/12)                        'ratio between each note = 12th root of 2
piano(1) = 27.5                                             'set first note...low A
FOR note = 2 TO 88
   piano(note) = piano(note - 1) * ratio     'multiply previous note by factor
NEXT note

FOR n = 0 TO 11 : READ noteName$(n) : NEXT                  'read note names
DATA G#, A, A#, B, C, C#, D, D#, E, F, F#, G

WIDTH , 7
FOR note = 1 TO 88                                              'play the scale
   octave = (note + 8) \12                              'calculate octave of note
   noteStr$ = noteName$(note MOD 12)                      'find its name in array
   PRINT noteStr$, octave, "piano(" note ")", piano(note)
   SOUND piano(note), 5
NEXT
```

FIGURE 18-11

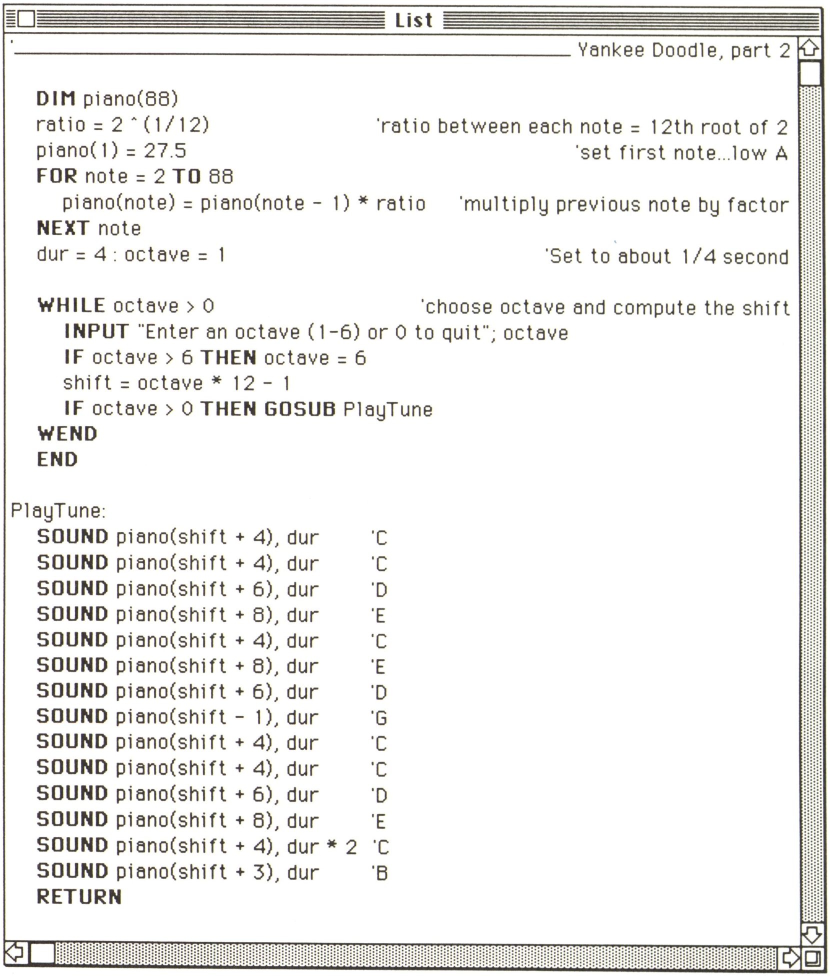
A program to compute the 88 musical notes of the piano

Note	Octave	Array Element	Frequency
A	0	piano(1)	27.5
A#	0	piano(2)	29.13524
B	0	piano(3)	30.86771
C	1	piano(4)	32.7032
C#	1	piano(5)	34.64783
D	1	piano(6)	36.7081
D#	1	piano(7)	38.89088
E	1	piano(8)	41.20345
F	1	piano(9)	43.65354
F#	1	piano(10)	46.24932
G	1	piano(11)	48.99945
G#	1	piano(12)	51.91311
A	1	piano(13)	55.00003
A#	1	piano(14)	58.2705
B	1	piano(15)	61.73545
C	2	piano(16)	65.40643
C#	2	piano(17)	69.29571
D	2	piano(18)	73.41624

FIGURE 18-12

A table of notes and their corresponding values in the piano *array*

As you might guess, using this array to determine the frequency parameter is much easier than consulting a table of frequencies. Figure 18-13 displays the "Yankee Doodle" program rewritten using the array. Notice the *shift* variable in the program. By changing the value of *shift* before we use the SOUND command, we can easily change the music produced. To change the octave, the program adds or subtracts a multiple of 12 to *shift*, based on the octave number entered by the user. We can even shift, or *transpose*, the key of the music by adding or subtracting any integer to *shift*. For instance, to change from the key of C to the key of C#, simply add 1 to *shift*.



```
'__________________________________________________________ Yankee Doodle, part 2

  DIM piano(88)
  ratio = 2 ^ (1/12)                      'ratio between each note = 12th root of 2
  piano(1) = 27.5                                           'set first note...low A
  FOR note = 2 TO 88
     piano(note) = piano(note - 1) * ratio    'multiply previous note by factor
  NEXT note
  dur = 4 : octave = 1                                     'Set to about 1/4 second

  WHILE octave > 0                              'choose octave and compute the shift
     INPUT "Enter an octave (1-6) or 0 to quit"; octave
     IF octave > 6 THEN octave = 6
     shift = octave * 12 - 1
     IF octave > 0 THEN GOSUB PlayTune
  WEND
  END

PlayTune:
  SOUND piano(shift + 4), dur      'C
  SOUND piano(shift + 4), dur      'C
  SOUND piano(shift + 6), dur      'D
  SOUND piano(shift + 8), dur      'E
  SOUND piano(shift + 4), dur      'C
  SOUND piano(shift + 8), dur      'E
  SOUND piano(shift + 6), dur      'D
  SOUND piano(shift - 1), dur      'G
  SOUND piano(shift + 4), dur      'C
  SOUND piano(shift + 4), dur      'C
  SOUND piano(shift + 6), dur      'D
  SOUND piano(shift + 8), dur      'E
  SOUND piano(shift + 4), dur * 2  'C
  SOUND piano(shift + 3), dur      'B
  RETURN
```

FIGURE 18-13

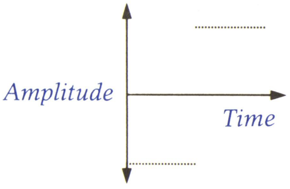
The "Yankee Doodle" program using an array to contain the notes

COMBINING MULTIPLE VOICES

So far we've seen how powerful Microsoft BASIC is for making sounds with one voice. But imagine the variety of sounds we could make using more than one voice at one time. This would enable us to create more complex sound effects that are richer and fuller in complexity than single-voice effects. Examples are simultaneous sounds (such as a car engine and a horn at the same time), or extensions of the single-voice effects (such as a glissando of four notes at a time). We can also create musical chords with up to four notes per chord.

You should be aware, however, that using multivoice sound will slow your BASIC programs down significantly. Although it is hard to measure, your programs may run as much as 50 percent slower if you use multivoice sounds. You must, unfortunately, decide whether the higher-quality sounds that you can create with multivoice sounds are worth the price in program-speed degradation. In many cases you can use multivoice sound by the careful placement of statements in your program where the delay that it causes is not a problem, such as in title screens or during periods where the user is simply reading the screen.

Along with multivoice sound comes the ability to control the actual *waveform* that makes each voice. Every musical instrument creates a certain characteristic *texture* of sound, which is related to its waveform. This is why a flute and a violin can play exactly the same note and yet sound different. The musical term for this phenomenon is *timbre.* The timbre of each instrument is unique because the shape and material of that particular instrument—whether string, reed, brass, and so on—produces minute variations within the pattern (or form) of each cycle (or wave) of sound. It is the *form* of each individual *wave* that we can define. A waveform played by the Macintosh is mathematically defined for the shape of the sound wave that is produced. The waveform of the default single-voice sound is a square wave that looks like this:



This shape was chosen by the creators of BASIC because it is very easy for a computer to produce.

In the following sections we'll learn how to use multivoice sounds and create waveforms that have timbres that sound more natural for music, as well as shapes that create weird effects.

Using the WAVE statement

To initialize multivoice sound, we must use the WAVE statement. (WAVE is also used to change the waveform of the sound generated; we will learn how to do this later in the chapter.) The WAVE command is usually given at the beginning of programs, in an initialization section. We cannot use multivoice sound until we have set up all the voices with a WAVE command.

The general syntax of the WAVE command is:

WAVE *Voice Number, Wave Shape, Phase*

Although the *Wave Shape* and *Phase* parameters are optional, not including the *Wave Shape* parameter turns off a particular voice.

The Voice Number parameter: Selecting the voice

The required *Voice Number* parameter is the same as that in the SOUND command: The range of values is from 0 to 3. Up to now, we have only been using the default value, voice 0. After we give the WAVE command a *Voice Number* other than 0, we can then use the *Voice Number* parameter of the SOUND command (the fourth parameter).

The Wave Shape parameter: Specifying the type of sound

The *Wave Shape* can be either the keyword SIN or the name of an integer array. The keyword *SIN* stands for *sine.* Consequently, we will usually use the SIN option because a sine wave is a pure flute-like tone, and this is the most pleasant tone for music. If we specify an array name, Microsoft BASIC will use the data in the array to define the shape of the wave that is produced. Using different wave shapes can make interesting sounds, and this is described at the end of this chapter. For now, we'll use the SIN option for the *WaveShape* parameter. For example, the series of commands to set up all four voices are:

```
WAVE 0, SIN
WAVE 1, SIN
WAVE 2, SIN
WAVE 3, SIN
```

Remember that the first voice number is 0, not 1.

The Phase parameter: Shifting the Wave Shape

The *Phase* parameter is used to change how BASIC accesses the array given as the *Wave Shape* parameter; this allows us to shift the shape of a wave within its waveform. This will be discussed later in the chapter. Since we are not yet using the parameters to specify custom wave shapes, our WAVE commands will only have the voice number and the SIN option.

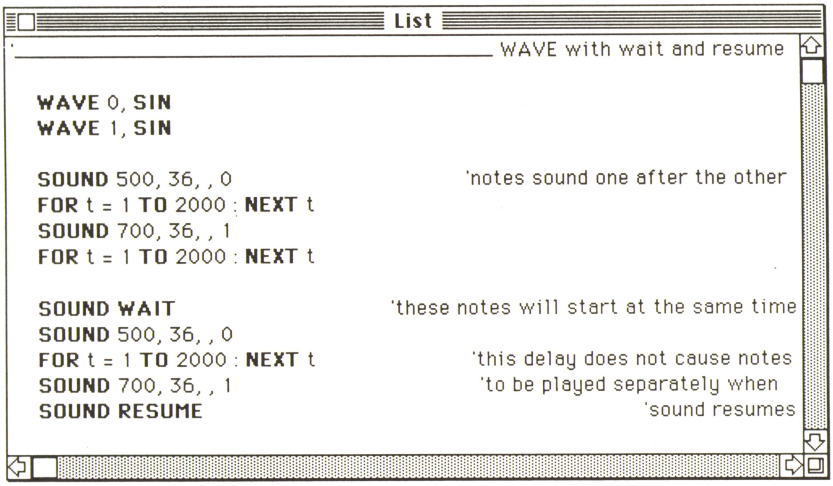
Making simultaneous sounds with SOUND WAIT and SOUND RESUME

Now that we know how to tell BASIC that we want to use multivoice sound, we're almost ready to start writing programs with it. However, there is still one small hurdle to overcome before you turn the Mac into a many-voiced choir. The example in Figure 18-14 demonstrates how to use SOUND WAIT and SOUND RESUME. The first part of the program plays one sound followed by another with a pause between them. The second part puts the same sounds between the SOUND WAIT and SOUND RESUME commands; this time the sounds are played at the same time.

The SOUND WAIT command tells BASIC to start saving sounds in a sound *queue*, which is like a "holding area" for sounds (this is analogous to the *queuing of events* we learned about in Chapter 14). When we've put all the sounds in the queue, the SOUND RESUME command causes BASIC to play the coordinated sounds. The size of the sound queue is limited, so you may get an "Out of memory" error if you have too many SOUND statements between the SOUND WAIT and SOUND RESUME commands.

Making advanced sound effects

The principles for generating multivoice sound effects are comparable to those for single-voice effects. The main advantage to using multivoice



```
'______________________________________________ WAVE with wait and resume

  WAVE 0, SIN
  WAVE 1, SIN

  SOUND 500, 36, , 0                            'notes sound one after the other
  FOR t = 1 TO 2000 : NEXT t
  SOUND 700, 36, , 1
  FOR t = 1 TO 2000 : NEXT t

  SOUND WAIT                            'these notes will start at the same time
  SOUND 500, 36, , 0
  FOR t = 1 TO 2000 : NEXT t                    'this delay does not cause notes
  SOUND 700, 36, , 1                             'to be played separately when
  SOUND RESUME                                                  'sound resumes
```

FIGURE 18-14

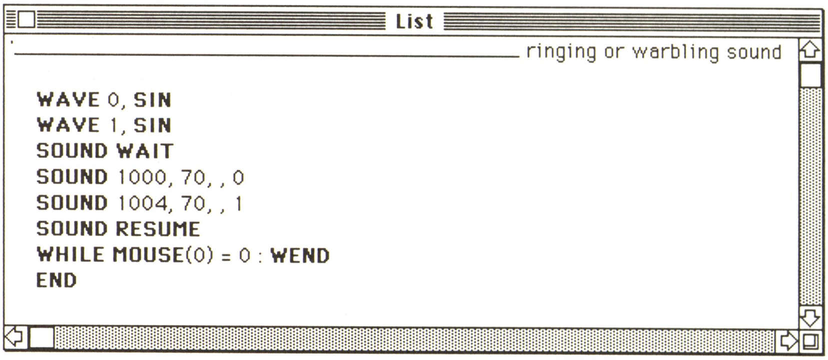
A program demonstrating the SOUND WAIT and SOUND RESUME commands effects is that we can create complex *harmonics,* which are sounds resulting from combining individual frequencies that are played together.

Let's suppose we created a flying spaceship and used single-voice sound to generate a single tone to simulate the spaceship's sound. With multivoice sound, we could use a combination of tones to create a more interesting humming sound instead:

```
SOUND 240, 50, , 0
SOUND 320, 50, , 1
```

The musical theory of harmonics and its study of the blending of separate sounds into pleasing or discordant sounds is well beyond the scope of this book. In fact, it's a subject that's challenged music theorists and mathematicians since Pythagoras. Most of us ordinary mortals have to experiment to find pleasing, harmonious sounds for our programs. Of course, we'll sometimes *want* jarring or strange sounds as well. By experimenting with the different frequencies, we can explore our way to the effect that's just right for a particular situation.

Combined sounds that are close in frequency often produce a "warbling" sound, often called a *beat frequency.* This warbling is especially useful for making sounds in space games or haunting sound effects, which you will hear if you type and run the program in Figure 18-15. Running BASIC on the Mac sometimes causes the disk to spin for often mysterious reasons. However, the WHILE...WEND loop at the end of the program will keep it from spinning while the sound is playing. The disk can cause bad effects on sounds, so a delay of the right point in a program is often useful to minimize any disk access when sounds are playing.



```
'___________________________________________ ringing or warbling sound

  WAVE 0, SIN
  WAVE 1, SIN
  SOUND WAIT
  SOUND 1000, 70, , 0
  SOUND 1004, 70, , 1
  SOUND RESUME
  WHILE MOUSE(0) = 0 : WEND
  END
```

FIGURE 18-15

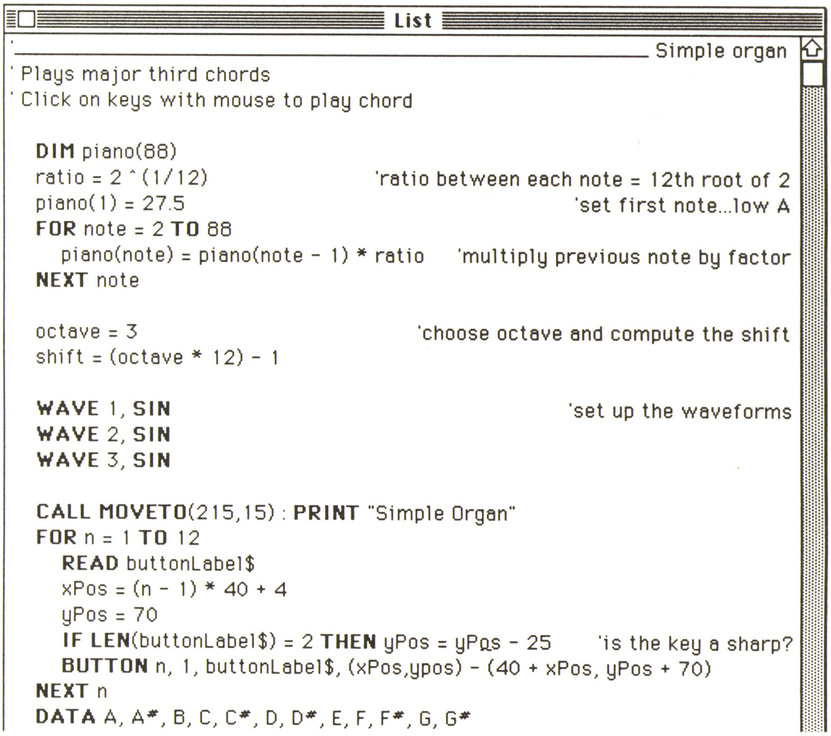
A program that generates a "warbling" sound

We can also use multivoice sounds to create beautiful frequency "slides." You'll hear what we mean if you modify the single-voiced sound-effect program in Figure 18-6 to use multivoice sound by adding the appropriate WAVE statements at the beginning of the program, and replacing the three SOUND statements with:

```
SOUND freq, dur, , 0
SOUND freq + 10, dur, , 1
SOUND freq + 20, dur, , 2
```

Programming musical chords

Writing programs that use musical chords is straightforward. If we use the *piano* array in Figure 18-11, we can use chord theory to form any type of chord. Figure 18-16 shows a program that plays major chords like an organ. The SOUND commands that generate the chord use the fact that a major chord is made up of three notes: the base note, the note four notes above the base, and the note seven notes above the base. Note how easy it is to create the sounds from the scale. The three SOUND statements use the *piano* array to generate their notes.



```
'
'______________________________________________________ Simple organ
' Plays major third chords
' Click on keys with mouse to play chord

  DIM piano(88)
  ratio = 2 ^ (1/12)                     'ratio between each note = 12th root of 2
  piano(1) = 27.5                                          'set first note...low A
  FOR note = 2 TO 88
     piano(note) = piano(note - 1) * ratio    'multiply previous note by factor
  NEXT note

  octave = 3                               'choose octave and compute the shift
  shift = (octave * 12) - 1

  WAVE 1, SIN                                               'set up the waveforms
  WAVE 2, SIN
  WAVE 3, SIN

  CALL MOVETO(215,15) : PRINT "Simple Organ"
  FOR n = 1 TO 12
     READ buttonLabel$
     xPos = (n - 1) * 40 + 4
     yPos = 70
     IF LEN(buttonLabel$) = 2 THEN yPos = yPos - 25        'is the key a sharp?
     BUTTON n, 1, buttonLabel$, (xPos,ypos) - (40 + xPos, yPos + 70)
  NEXT n
  DATA A, A*, B, C, C*, D, D*, E, F, F*, G, G*
```

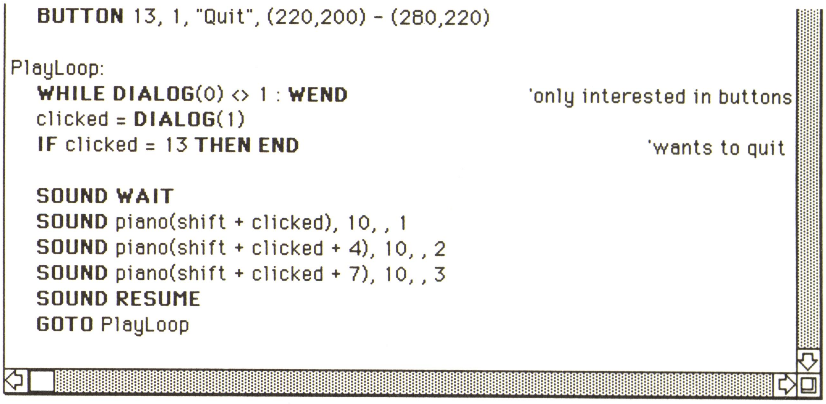
FIGURE 18-16

A program that simulates a simple chord organ

(continued)

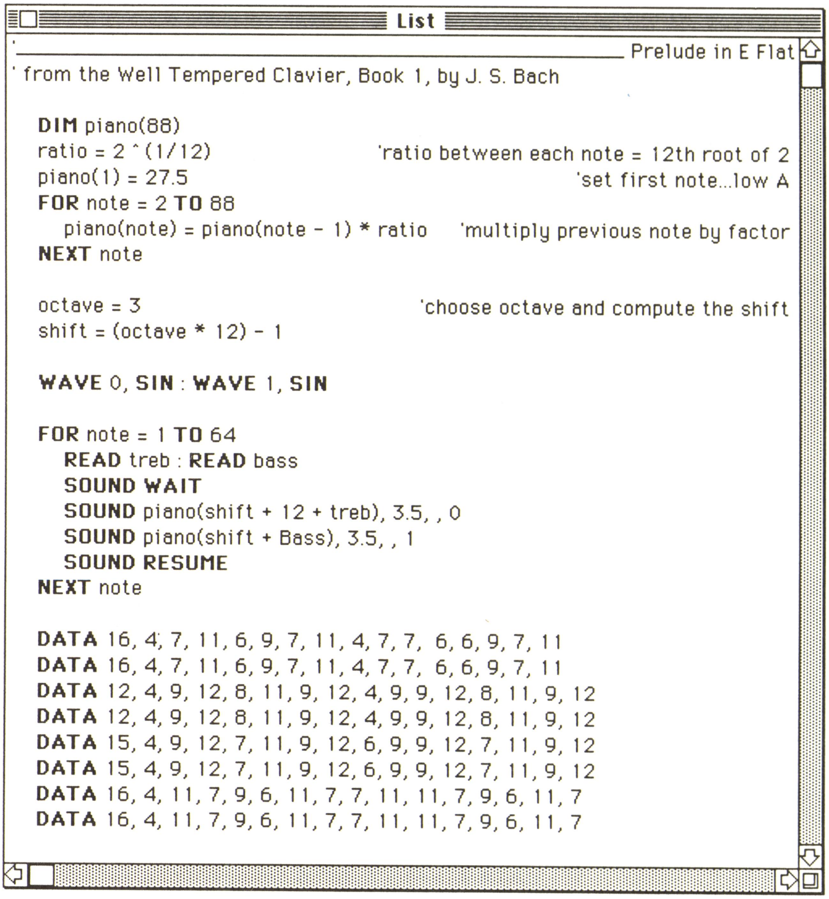
FIGURE 18-16

(continued)



```
   BUTTON 13, 1, "Quit", (220,200) - (280,220)

PlayLoop:
   WHILE DIALOG(0) <> 1 : WEND                       'only interested in buttons
   clicked = DIALOG(1)
   IF clicked = 13 THEN END                               'wants to quit

   SOUND WAIT
   SOUND piano(shift + clicked), 10, , 1
   SOUND piano(shift + clicked + 4), 10, , 2
   SOUND piano(shift + clicked + 7), 10, , 3
   SOUND RESUME
   GOTO PlayLoop
```

We can use the same principles to play musical pieces. Figure 18-17 shows a program that plays the first few bars of "Prelude in E Flat"



```
'_____________________________________________________ Prelude in E Flat
' from the Well Tempered Clavier, Book 1, by J. S. Bach

   DIM piano(88)
   ratio = 2 ^ (1/12)                'ratio between each note = 12th root of 2
   piano(1) = 27.5                                    'set first note...low A
   FOR note = 2 TO 88
      piano(note) = piano(note - 1) * ratio     'multiply previous note by factor
   NEXT note

   octave = 3                             'choose octave and compute the shift
   shift = (octave * 12) - 1

   WAVE 0, SIN : WAVE 1, SIN

   FOR note = 1 TO 64
      READ treb : READ bass
      SOUND WAIT
      SOUND piano(shift + 12 + treb), 3.5, , 0
      SOUND piano(shift + Bass), 3.5, , 1
      SOUND RESUME
   NEXT note

   DATA 16, 4, 7, 11, 6, 9, 7, 11, 4, 7, 7,  6, 6, 9, 7, 11
   DATA 16, 4, 7, 11, 6, 9, 7, 11, 4, 7, 7,  6, 6, 9, 7, 11
   DATA 12, 4, 9, 12, 8, 11, 9, 12, 4, 9, 9, 12, 8, 11, 9, 12
   DATA 12, 4, 9, 12, 8, 11, 9, 12, 4, 9, 9, 12, 8, 11, 9, 12
   DATA 15, 4, 9, 12, 7, 11, 9, 12, 6, 9, 9, 12, 7, 11, 9, 12
   DATA 15, 4, 9, 12, 7, 11, 9, 12, 6, 9, 9, 12, 7, 11, 9, 12
   DATA 16, 4, 11, 7, 9, 6, 11, 7, 7, 11, 11, 7, 9, 6, 11, 7
   DATA 16, 4, 11, 7, 9, 6, 11, 7, 7, 11, 11, 7, 9, 6, 11, 7
```

FIGURE 18-17

Music by Bach, played by the Well-Tempered MacClavier from *The Well-Tempered Clavier* by J. S. Bach. The notes were transcribed directly from the sheet music by first listing the notes (A, D#, and so on), then converting the notes to the numbers in the *piano* array.

CHANGING THE WAVEFORM

Each sound has a characteristic shape of the wave that produces the sound. Thus, changing the shape of the waveform also changes the texture of the sound. "Pure" sounds are generated with a sine wave. If we use single-voice sound without first giving the WAVE command, BASIC generates a square wave; if we use the SIN option in the WAVE command, we get a wave shaped like a sine curve. We can use the WAVE command to specify a different shape for the sound wave for each voice of a multivoiced sound.

Creating the waveform array

To change the shape of the wave, we specify an integer array of 256 elements in the WAVE command. Each element in the array must contain a value between −128 to 127. The 256 elements of the array represent one cycle of sound. The values that are stored in the array determine the shape of the wave. The array must be filled with the values before we give the WAVE command.

Remember that the general syntax of the WAVE command is:

WAVE *Voice Number, Wave Shape, Phase*

We've used the SIN option for the *Wave Shape* parameter up to now. Instead of *SIN,* we can change the shape of the wave by giving the name of the array that holds the values, such as:

```
WAVE 1, musicArray%
```

This array can also be a multidimensional array of 256 by *n* elements, where *n* is an index defining which waveform we wish to use. This technique is similar to that described in the previous chapter for using an indexed array to change shapes defined with the PUT statement.

The *Phase* option tells BASIC to start reading the array at a point different from the first element. Using *Phase* is generally useful only if you are defining two sounds with the same array, since it changes the way in which BASIC sets up the sound. The *Phase* option specifies the position in the array to use as the first value of the waveform.

For example, if the array *waveTri%* is to define a triangle wave, the statements:

```
DIM waveTri%(255)
FOR n = 0 TO 127
   waveTri%(n) = n * 2 - 127
   waveTri%(n + 128) = 127 - n * 2
NEXT n
```

are used to fill the array with the appropriate values to generate this wave shape. The two statements:

```
WAVE 0, waveTri%
WAVE 1, waveTri%
```

would produce waves that were in-sync with each other.

If we included a *Phase* parameter in one of the WAVE commands, the waves would be out-of-sync. Let's change the second WAVE command to shift the triangle wave of voice 1 by half a cycle:

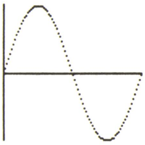
```
WAVE 1, waveTri%, 127
```

The contents of the wave array can be anything we want, but there are a few shapes that are common for sound waves. The square wave is common for single-voice sounds, since it is the easiest for a computer to produce. A sine wave produces a better-sounding tone than a square wave (especially when more than one voice is used), which is why Microsoft gave that as the default for multivoice sound.

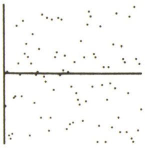
Since sine waves produce pure sounds, variations on sine waves produce sounds that are close to sine waves but with different qualities. Another common wave shape used in music synthesizers is a triangle wave, which rises and falls linearly. Figure 18-18 shows a collection of some useful sounds and the BASIC statements that generate them.

FIGURE 18-18

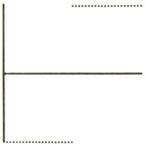
"Recipes" for some common wave shapes



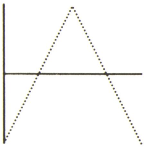
```
DIM waveSine%(255)                    'sine wave...flute-like
FOR n = 0 TO 255
   waveSine%(n) = 127 * SIN((n/256) * 6.2838)
NEXT n
```



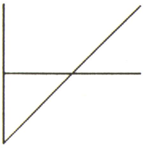
```
DIM waveHiss%(255)              'random noise...hissing, buzzing
FOR n = 0 TO 255
   waveHiss%(n) = RND * 256 - 128
NEXT n
```



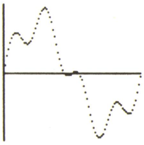
```
DIM waveSqr%(255)                  'square wave...computer-like
FOR n = 0 TO 127
   waveSqr%(n) = -128
   waveSqr%(255 - n) = 127
NEXT n
```



```
DIM waveTri%(255)                     'triangle wave...oboe-like
FOR n = 0 TO 127
   waveTri%(n) = (n * 2) - 128
   waveTri%(255 - n) = (n * 2) - 128
NEXT n
```



```
DIM waveSaw%(255)                  'sawtooth wave...violin-like
FOR n = 0 TO 255
   waveSaw%(n) = n - 128
NEXT n
```

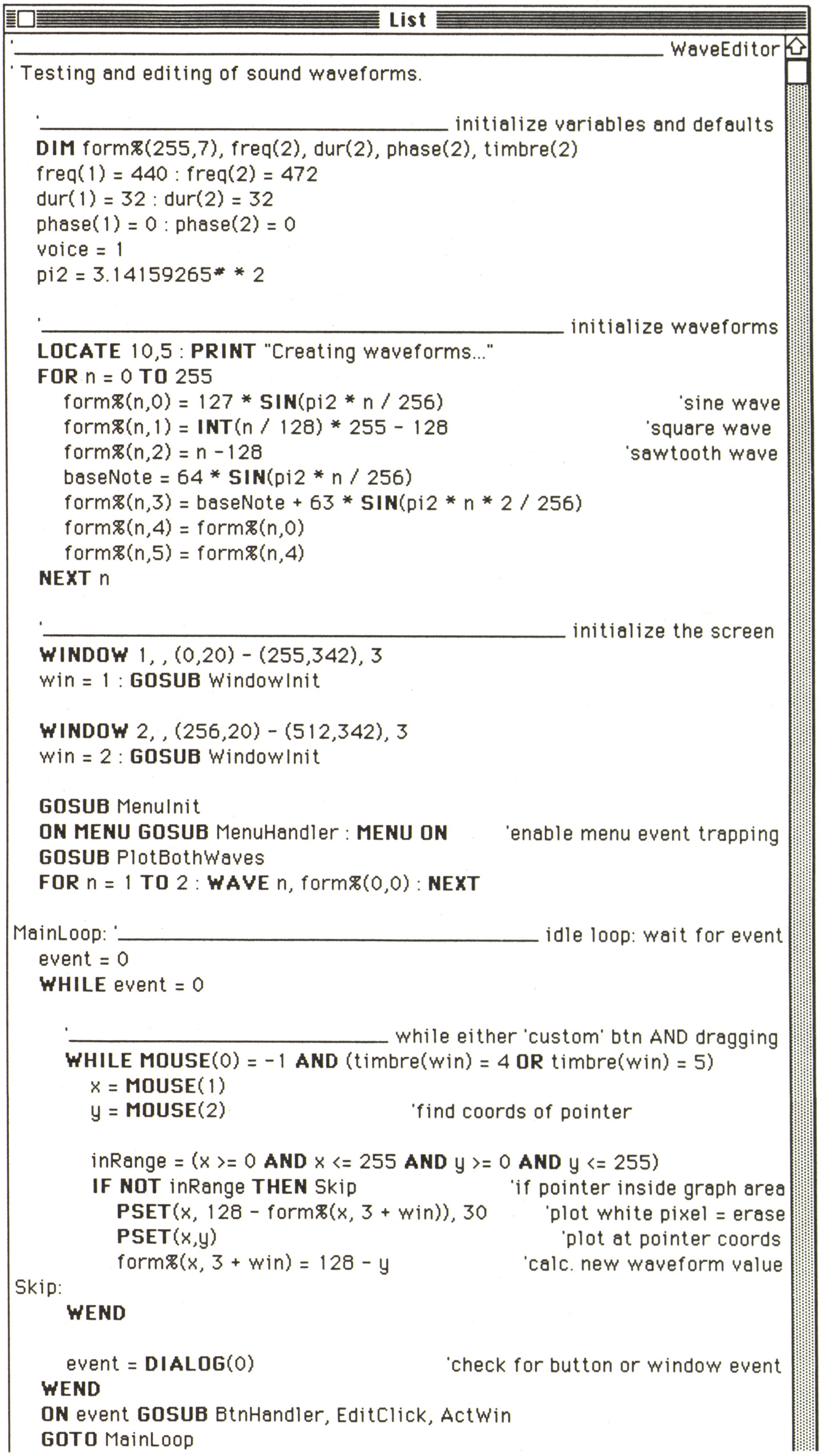


```
DIM waveOrgan%(255)         'addition of sine waves...organ-like
FOR n = 0 TO 255
   majorVoice = 96 * SIN((n/256) * 6.2838)
   minorVoice = 32 * SIN((n/64) * 6.2838)      '1/3 vol, 4x freq
   waveOrgan%(n) = majorVoice + minorVoice      '2 voices into 1
NEXT n
```

Figure 18-19, which begins on page 482, is the last program in the chapter. It combines all of the concepts that we've learned up to now about the SOUND command, and lets us experiment with changing the shapes of the waves in two voices. The Output window for the program is shown in Figure 18-20 on page 485.

FIGURE 18-19

The waveform-tester program

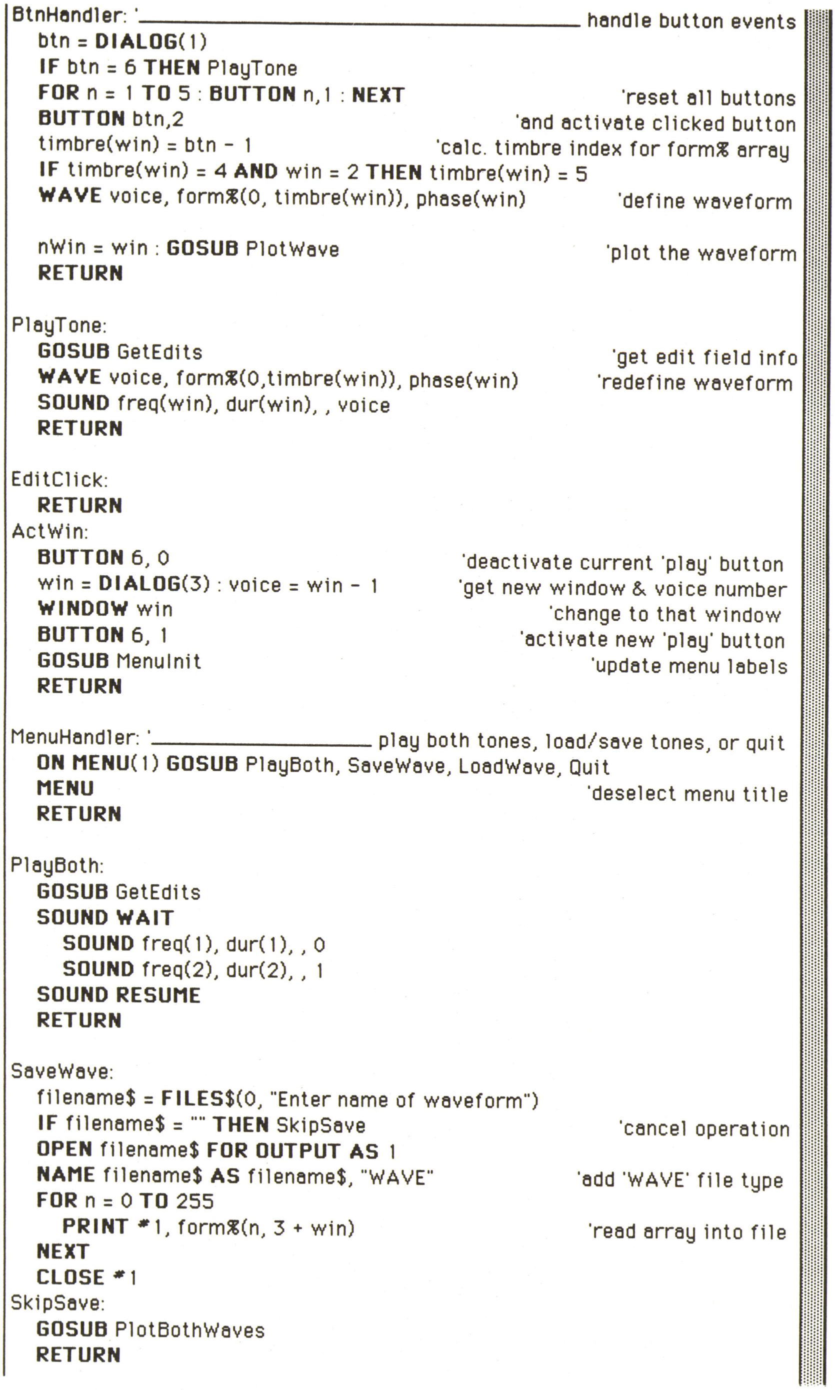


```
'_____________________________________________________________ WaveEditor
' Testing and editing of sound waveforms.

   '________________________________________ initialize variables and defaults
   DIM form%(255,7), freq(2), dur(2), phase(2), timbre(2)
   freq(1) = 440 : freq(2) = 472
   dur(1) = 32 : dur(2) = 32
   phase(1) = 0 : phase(2) = 0
   voice = 1
   pi2 = 3.14159265# * 2

   '_____________________________________________________ initialize waveforms
   LOCATE 10,5 : PRINT "Creating waveforms..."
   FOR n = 0 TO 255
      form%(n,0) = 127 * SIN(pi2 * n / 256)                        'sine wave
      form%(n,1) = INT(n / 128) * 255 - 128                      'square wave
      form%(n,2) = n -128                                       'sawtooth wave
      baseNote = 64 * SIN(pi2 * n / 256)
      form%(n,3) = baseNote + 63 * SIN(pi2 * n * 2 / 256)
      form%(n,4) = form%(n,0)
      form%(n,5) = form%(n,4)
   NEXT n

   '_____________________________________________________ initialize the screen
   WINDOW 1, , (0,20) - (255,342), 3
   win = 1 : GOSUB WindowInit

   WINDOW 2, , (256,20) - (512,342), 3
   win = 2 : GOSUB WindowInit

   GOSUB MenuInit
   ON MENU GOSUB MenuHandler : MENU ON         'enable menu event trapping
   GOSUB PlotBothWaves
   FOR n = 1 TO 2 : WAVE n, form%(0,0) : NEXT

MainLoop: '_____________________________________________ idle loop: wait for event
   event = 0
   WHILE event = 0

      '________________________________ while either 'custom' btn AND dragging
      WHILE MOUSE(0) = -1 AND (timbre(win) = 4 OR timbre(win) = 5)
         x = MOUSE(1)
         y = MOUSE(2)                             'find coords of pointer

         inRange = (x >= 0 AND x <= 255 AND y >= 0 AND y <= 255)
         IF NOT inRange THEN Skip                      'if pointer inside graph area
            PSET(x, 128 - form%(x, 3 + win)), 30          'plot white pixel = erase
            PSET(x,y)                                       'plot at pointer coords
            form%(x, 3 + win) = 128 - y                   'calc. new waveform value
Skip:
      WEND

      event = DIALOG(0)                          'check for button or window event
   WEND
   ON event GOSUB BtnHandler, EditClick, ActWin
   GOTO MainLoop
```

(continued)

FIGURE 18-19

(continued)



```
BtnHandler: '________________________________ handle button events
   btn = DIALOG(1)
   IF btn = 6 THEN PlayTone
   FOR n = 1 TO 5 : BUTTON n,1 : NEXT                'reset all buttons
   BUTTON btn,2                                 'and activate clicked button
   timbre(win) = btn - 1                'calc. timbre index for form% array
   IF timbre(win) = 4 AND win = 2 THEN timbre(win) = 5
   WAVE voice, form%(0, timbre(win)), phase(win)       'define waveform

   nWin = win : GOSUB PlotWave                        'plot the waveform
   RETURN

PlayTone:
   GOSUB GetEdits                                     'get edit field info
   WAVE voice, form%(0,timbre(win)), phase(win)      'redefine waveform
   SOUND freq(win), dur(win), , voice
   RETURN

EditClick:
   RETURN
ActWin:
   BUTTON 6, 0                          'deactivate current 'play' button
   win = DIALOG(3) : voice = win - 1    'get new window & voice number
   WINDOW win                                  'change to that window
   BUTTON 6, 1                               'activate new 'play' button
   GOSUB MenuInit                               'update menu labels
   RETURN

MenuHandler: '______________ play both tones, load/save tones, or quit
   ON MENU(1) GOSUB PlayBoth, SaveWave, LoadWave, Quit
   MENU                                         'deselect menu title
   RETURN

PlayBoth:
   GOSUB GetEdits
   SOUND WAIT
      SOUND freq(1), dur(1), , 0
      SOUND freq(2), dur(2), , 1
   SOUND RESUME
   RETURN

SaveWave:
   filename$ = FILES$(0, "Enter name of waveform")
   IF filename$ = "" THEN SkipSave                    'cancel operation
   OPEN filename$ FOR OUTPUT AS 1
   NAME filename$ AS filename$, "WAVE"            'add 'WAVE' file type
   FOR n = 0 TO 255
      PRINT #1, form%(n, 3 + win)                   'read array into file
   NEXT
   CLOSE #1
SkipSave:
   GOSUB PlotBothWaves
   RETURN
```

(continued)

FIGURE 18-19

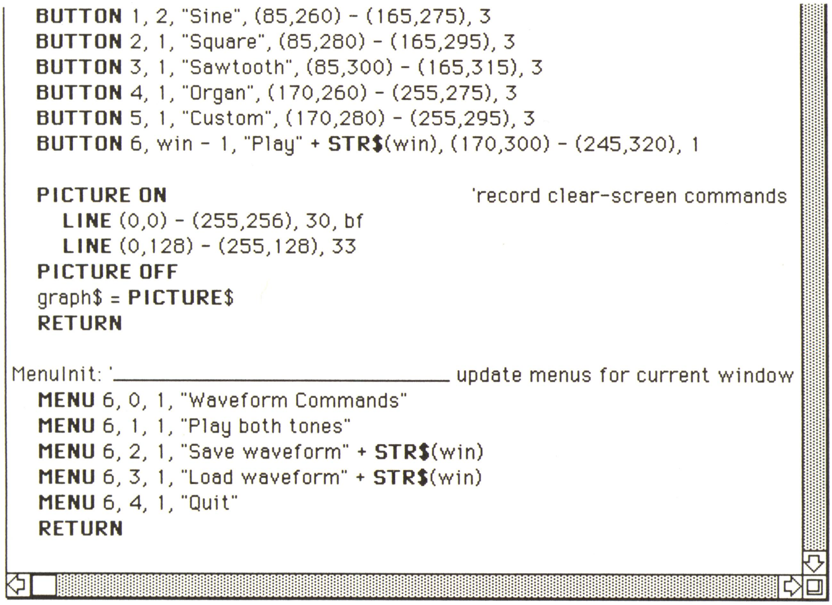
(continued)

```
LoadWave:
   filename$ = FILES$(1, "WAVE")
   IF filename$ = "" THEN SkipLoad                           'cancel operation
   OPEN filename$ FOR INPUT AS 1
   FOR n = 0 TO 255
      INPUT #1, form%(n, 3 + win)               'load file into appropriate array
   NEXT
   CLOSE #1
SkipLoad:
   GOSUB PlotBothWaves
   RETURN

Quit:
   MENU RESET
   FOR win = 1 TO 2
      WINDOW win
      FOR n = 1 TO 6 : BUTTON CLOSE n : NEXT                      'close buttons
      FOR n = 1 TO 3 : EDIT FIELD CLOSE n : NEXT              'close edit fields
      WINDOW CLOSE win
   NEXT win
   END

PlotBothWaves: '____________________ draw waveforms in pertinent window
   FOR nWin = 1 TO 2
      WINDOW nWin
      GOSUB PlotWave
   NEXT
   WINDOW win
   RETURN

PlotWave:
   PICTURE , graph$                                        'clear & draw graph
   FOR n = 0 TO 255                           'plot waveform for chosen timbre
      PSET (n, 128 - form%(n,timbre(nWin))), 33                'plot each point
   NEXT n
   RETURN

GetEdits: '____________________ get contents of edit fields for each window
   FOR nWin = 1 TO 2
      WINDOW nWin
      freq(nWin) = VAL(EDIT$(1))
      dur(nWin) = VAL(EDIT$(2))
      IF dur(nWin) > 77 THEN dur(nWin) = 77
      phase(nWin) = VAL(EDIT$(3))
   NEXT nWin
   WINDOW win                                  'change back to current window
   RETURN

WindowInit: '___________________________ same setup for each window
   CALL MOVETO(2,272) : PRINT "freq."
   CALL MOVETO(2,292) : PRINT "dur."
   CALL MOVETO(2,312) : PRINT "phase";
   EDIT FIELD 1, STR$(freq(win)), (40,260) - (80,275), 1, 1
   EDIT FIELD 2, STR$(dur(win)), (40,280) - (80,295), 1, 1
   EDIT FIELD 3, STR$(phase(win)), (40,300) - (80,315), 1, 1
```

(continued)

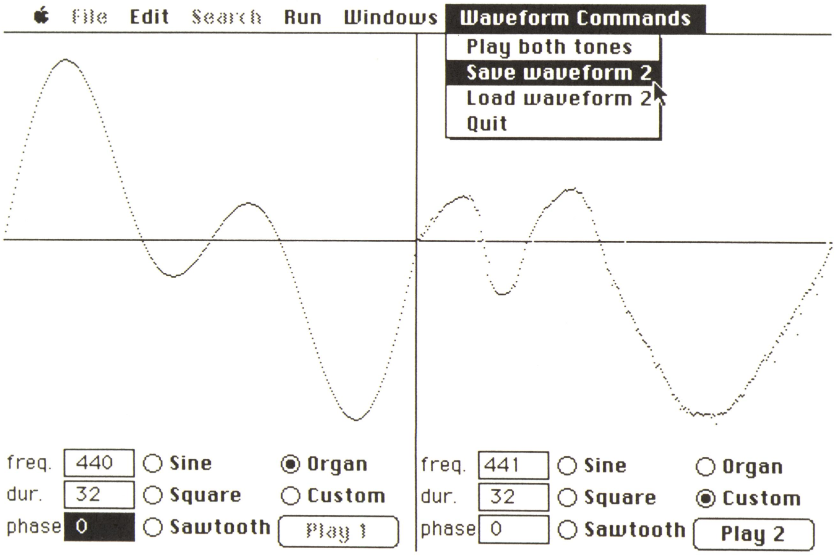
FIGURE 18-19

(continued)



```
   BUTTON 1, 2, "Sine", (85,260) - (165,275), 3
   BUTTON 2, 1, "Square", (85,280) - (165,295), 3
   BUTTON 3, 1, "Sawtooth", (85,300) - (165,315), 3
   BUTTON 4, 1, "Organ", (170,260) - (255,275), 3
   BUTTON 5, 1, "Custom", (170,280) - (255,295), 3
   BUTTON 6, win - 1, "Play" + STR$(win), (170,300) - (245,320), 1

   PICTURE ON                                    'record clear-screen commands
      LINE (0,0) - (255,256), 30, bf
      LINE (0,128) - (255,128), 33
   PICTURE OFF
   graph$ = PICTURE$
   RETURN

MenuInit: '__________________________ update menus for current window
   MENU 6, 0, 1, "Waveform Commands"
   MENU 6, 1, 1, "Play both tones"
   MENU 6, 2, 1, "Save waveform" + STR$(win)
   MENU 6, 3, 1, "Load waveform" + STR$(win)
   MENU 6, 4, 1, "Quit"
   RETURN
```

When you run the program, you can choose the shape of either voice with the buttons at the bottom of the screen. You can also design your own custom waveforms by drawing the shape on the screen with the mouse. To hear the sound that you define, click the appropriate "play" button. The edit boxes allow you to change the frequency, duration, and phase of the sound. The menu options Load and Save allow the storage of these


File
Edit
Search
Run
Windows
Waveform Commands
Play both tones
Save waveform 2
Load waveform 2
Quit
○ Sine
○ Square
○ Sawtooth
● Organ
○ Custom
Play 1
○ Sine
○ Square
○ Sawtooth
○ Organ
● Custom
Play 2

freq. 440
dur. 32
phase 0

freq. 441
dur. 32
phase 0

FIGURE 18-20

The Output window of the waveform-tester program custom waveforms on disk as sequential files, so you can transfer them into your own programs as needed.

REFRESH-YOURSELF WINDOW

As you can see, Microsoft BASIC gives us many interesting and powerful options for using sound in our programs. Remember, however, that multi-voice sound will often significantly slow the performance of a program, and probably cause more disk accesses on a 128K Macintosh.

When you consider how sound can be used in your programs, think about some of the programs we've presented earlier in this book, as well as the programs you may have written on your own. For each program, consider:

- Where could music be added to the program to make it more interesting?
- Could a new sound effect enhance a specific part of a game or an interactive application?
- Would using sound in a certain place detract from the overall purpose of the program?

Of course, you must also weigh the speed degradation and greater overhead of more programming steps when you add SOUND commands.

We mentioned earlier in this chapter that music is an art and a science that has challenged some of the best minds throughout history. Although this chapter has covered the basic principles of composing sound and music for the Macintosh with Microsoft BASIC, it has necessarily only scratched the surface of this complex subject. If you wish to further pursue the creation of music with the Macintosh, we suggest that you consult *BASIC Primer for the IBM PC and XT,* by Bernd Enders and Bob Petersen (Plume/Waite, New American Library, 1984), which contains an expanded discussion of music as it relates to Microsoft BASIC.

THE LAST END STATEMENT

We've finally come to the end of our Microsoft BASIC story. If reading this book is your first experience with programming, you should congratulate yourself on absorbing an enormous amount of complicated material. You may not have learned every detail about BASIC's operation, but you've been introduced to all the major areas. At this point you should—in theory at least—be able to sit down and write almost any program you want.

However, in programming, as in most endeavors, practice is the key to success. This book has given you the tools to do the job; now it's up to you to put them to work. Don't be discouraged if your first attempt to write a major program is not the piece of cake you expected. Just as you shouldn't try to build a whole house if you've never driven a nail before, so you should write smaller, more manageable programs until you're ready to tackle the big ones.

If you're already a programmer and have been reading this book to find out specifically about Microsoft BASIC for the Macintosh, then you must be impressed by how much more versatile and powerful this version of BASIC is than any previous implementation. Not only does it contain dozens of new statements for directing program control, it also gives you the power to control all those aspects of the Macintosh that make this marvelous computer such a revolutionary success: the mouse, menus, windows, and dialog boxes. With these tools at your disposal you will be able to write true "Mac-like" programs, which are at this point the cutting edge of the computer revolution.

INDEX

A

ABS function, 157
Addition, 33–34
Animation
  array sizes in, 442–44
  images in
    enlarging, 444–46
    erasing, 435–36
    MacPaint, 451–56
    moving, 434–46
    multiple, 446–51
    rotating, 446–51
  interacting with background, 436–42
Apple menu, 22
Arrays, 178–80
  accessing, 178, 188
  arithmetic operations with, 182–85
  elements, 178–79
  erasing, 191
  memory addresses of, 370
  multidimensional, 186–90
  organization of, 178–79
  size of, 180, 187, 191
  string, 185–86, 189–90
  two-dimensional, 448
ASC function, 212–13
ASCII codes, 209–13
  characters represented by, 209–12
  converting strings to, 214–15
  fonts, 348–49
Assignment statements, 98–99
  in array elements, 181
  versus logical expressions, 129–31

B

BACKPAT statement, 376
Backspace key, 51–53, 56, 86
BASIC
  functions, 30, 135–36
  interfacing to MacPaint, 451–56

BASIC *(continued)*
  labels, 139
  program disk
    backing up, 5, 7–8, 11–12
    replacing, 4
    upgrading, 4
  quitting, 22, 66
  registration card, 4
  restarting with backup, 13
  starting, 16, 67
  statements, 30, 135
  strings, 32, 40
  versions of
    binary, 6, 13–16
    decimal, 6–7, 13, 16
BEEP statement, 41
Box options with LINE, 80–81
BUTTON statement, 282–88
  Rectangle parameter, 284
  Title parameter, 284
BUTTON CLOSE statement, 298–301
Buttons, 280–82
  Cancel, 62, 66
  check boxes, 281
  creating, 282–88
  Drive, 62
  programming
    architecture, 295–96
    multiple, 285–86
    options, 296–98
    using IF...THEN statements, 293–94
    using ON...GOSUB statement, 294–95
  push, 281
  radio, 281

C

CALL statement, 341–42
Characters
  ASCII codes, 212–13
  changing to uppercase, 221

Characters *(continued)*
  graphic, 350–52
  repeating, 222–23
  searching for, 215–16
CHR$ function, 166, 210–11
CHR$ statement, 350–52
CIRCLE statement, 81–82, 124, 126
Clipboard
  retrieving text from, 57
  space available on, 57
  storing text in, 56
CLOSE statement, 397–98
  File Number parameter, 398
CLS statement, 33
Command window, 17, 27–28
  activating, 20
  expanding, 20
  making entries in, 28–29
  moving, 20
  resizing, 20
Commands
  keyboard, 23
Coordinate system, 73–74
  absolute, 309
  relative, 309–10
Copy command, 56, 58–59
Cut command, 56
CVD function, 422
CVI function, 422
CVS function, 422–23

D

DATA statement, 192–97
  creating menus using, 270–71
  data pointer, 194–95
    resetting, 197
DATE$ function, 40–41
DEFINT statement, 367
Dialog boxes
  copying disks, 12
  creating, 326–32
    buttons, 329–30
  initialization, 10
  naming disks, 11
  Open, 65–66
  saving programs, 61–62, 64
DIALOG function, 289–92, 318–20
DIALOG OFF statement, 322
DIALOG ON statement, 322
DIM statement, 180–81
  multiple array names in, 183
  in two-dimensional arrays, 187
Disks
  copying, 11–12
  dedicated, 7
  destination, 8
  dominant, 13
  drives, changing, 62
  icons, 6
  initializing, 9–10
  list of programs on, 65
  naming, 10
  source, 8
  windows, 8
  write-protecting, 5
Division, 34

E

EDIT FIELD statement, 326
  Default String parameter, 327
  ID parameter, 327
  Rectangle parameter, 327
Edit menu, 23
  editing commands in, 56–59
Eject button, 62
END statement, 113
EOF function, 401–3, 421–22
ERASE statement, 191
ERASERECT statement, 384
Error messages, 29–30
  DATA statement, 195
  FOR...NEXT loop, 125
  GOSUB statement, 162
  INPUT statement, 102
  ON GOSUB, 174
  SOUND statement, 475

F

FIELD statement, 416–17
  Field Width parameter, 416
  File Number parameter, 416
  String Variable parameter, 416
File menu, 22

Files
  defined, 389–92
  icons, 61
  naming, 406–10
  opening, 65–66
  random-access, 392–94, 413–14
    accessing, 425–26
    adding data to, 423–25
    index table in, 427–28
    opening, 415–18
    reading sequentially from, 420–23
    record lengths of, 413–14
    writing data to, 418–19
  sequential-access, 392–94
    adding data to, 400–401
    creating, 394–99
    finding the end of, 401–3
    reading data from, 399–400
    summary of, 412
FILES$ function, 406–10
  Prompt String parameter, 407
FILLRECT statement, 383
Font Mover program, 345–46
Fonts. *See also* TEXTFACE, TEXTFONT, TEXTMODE, TEXTSIZE
  accessing, 343–44
  ASCII codes for, 211–12
  available on BASIC disk, 344–46
  Cairo, 347–50
  copying, 345–46
  Geneva, 85, 166, 347–50
  point size of, 342
  proportional, 85
  scaling, 90, 342
  sizes, 89–90, 343–44
FOR...NEXT loops, 118–28
  in arrays, 182
  counter variable in, 119, 122
  creating menus using, 269–70
  STEP option, 122–23
  using a variable limit in, 127–28
FRAMERECT statement, 381
Functions
  ABS, 157
  ASC, 212–13
  CVD, 422
  CVI, 422

Functions *(continued)*
  CVS, 422–23
  DATE$, 40–41
  DIALOG, 289–92
  EOF, 401–3, 421–22
  FILES$, 406–10
  INKEY$, 243–45
  INSTR, 215–18
  INT, 150–51
  LEFT$, 204–7
  LEN, 207–8
  LOF, 424
  MENU, 256–59
  MID$, 206–7
  MKD$, 419
  MKI$, 419
  MKS$, 419
  MOD, 190
  RIGHT$, 204–7
  RND, 150–51
  SPACE$, 223
  SPC, 223
  STR$, 214–15
  STRING$, 222–23
  TIME$, 40–41
  UCASE$, 221
  VAL, 214–15

G

GET statement, 432–34
GET# statement, 421
  Record Number parameter, 421
GOSUB statement, 160–71
GOTO statement, 118, 138–40
Graphics. *See also* QuickDraw
  animated character, 165–71
  boxes, 80–81
  characters, 350–52
  circles, 81–82
    concentric, 123, 126
    rows of, 123–24
  coordinate system, 137
  lines, 78–79
  logical operators in statements, 154–55
  mixing text with, 91, 352–53
  ovals, 384–85

Graphics *(continued)*
  patterns
    background, 376
    creating, 373–75
  rectangles, 381–86

I

I-beam pointer. *See* Insertion point
Icons, 6, 66–67
  application, 66
  file, 61
  program, 66
IF...THEN statement, 144–48
  shortening, 152–53
  using with buttons, 293–94
IF...THEN...ELSE statement, 150
INKEY$ function, 243–45
INPUT statement, 100–105
  error message, 102
  prompt, 103
INPUT# statement, 399–400
  Expression List parameter, 400
  File Number parameter, 400
Insertion point
  positioning, 55
    in List window, 50–51
    in pasted material, 57
  selecting text with, 51–54, 56
INSTR function, 215–18
INT function, 150–51
Integers. *See* INT function
INVERTRECT statement, 383–84

K

Keyboard commands, 23

L

Labels, 138–40
LEFT$ function, 204–7
LEN function, 207–8
LINE statement, 78–81, 145
LINE INPUT statement, 218–19
List window
  clearing, 46, 53, 86
  closing, 111
List window *(continued)*
  deleting text from, 50–52
  displaying results of, 55
  expanding, 19–20, 110
  inserting new lines of text in, 55
  locking, 63
  moving, 19
  positioning insertion point in, 50–51
  programming in, 46–68
  replacing text in, 53
  resizing, 18–19
  scrolling through, 17–18
  using more than one, 20
LOCATE statement, 84–87
LOF function, 424
Logical expressions, 144–45
Logical operators, 151–52
  AND, 152–53
  in graphics, 154–55
  in IF...THEN statements, 152–53
  OR, 152–53
  in WHILE...WEND loops, 153–54
Loops, 117–18
  FOR...NEXT, 118–28
  GOTO, 139–40
  indentation of, 125
  infinite, 403
  nested, 123–26
  tracing through, 120–21
  WHILE...WEND, 128–34
LSET statement, 418

M

Macintosh
  clock, 40
  Control Panel
    copy patterns from, 375
    volume setting, 461
  desk accessories, 22, 40
  desktop, 6
    assigning patterns to, 368
    loading programs from, 67
Mathematical operations, 33–36
  using ASCII codes in, 214–15
Menu bar, 252–53
  controlling, 261–62
MENU function, 256–59

MENU statement, 253–55
Menus, 252
  Apple, 22
  choosing from, 9
  creating, 253–55
  dimmed, 22
  Edit, 23
  File, 22
  programming, 256–59
    architecture, 259–60
    check marks, 272–75
    dimmed, 275–76
    multiple, 264–69
    using DATA and READ statements, 270–71
    using FOR...NEXT statement, 269–70
  replacing BASIC, 271–72
  Run, 24
  Search, 23
  Windows, 24
MID$ function, 206–7
MKD$ function, 419
MKI$ function, 419
MKS$ function, 419
MOD function, 190
Mouse
  button
    click rate, setting in Control Panel, 232
    event trapping, 237–42
    status of, 135–36, 230–32
  calculating array sizes with, 443–44
  dragging
    drawing while, 234–37
    programming, 232–34
  programs
    drawing, 145–47
    pointer location, 136–37, 154–55
MOUSE GOSUB statement, 237
MOUSE OFF statement, 240–41
MOUSE ON statement, 240
MOUSE(0) function, 228–29, 231–32
MOVETO statement, 361–62
Multiplication, 34
Music. *See also* Sound
  chords, 473, 476–78
  frequencies, table of, 468–69
Music *(continued)*
  notes, 466–72
    piano, 470–72
    raising/lowering by octaves, 469–72
  scales, 466, 468–70
  timbre, 473

N

New command, 46

O

ON...GOSUB statement, 172–74
  with buttons, 294–95
  and menu events, 262–63
ON...GOTO statement, 262–63, 265–66
  with buttons, 294–95
ON MOUSE GOSUB, 240
Open... command, 65–66
OPEN statement, 399–400
  File-Buffer Size parameter, 416
  File Name parameter, 395
  File Number parameter, 396
  Mode parameter, 395–96
  in random-access files, 415–16
OPTION BASE statement, 190–91, 448
Output window, 17, 28
  clearing, 33
  columns in, 36–37
  coordinate system on, 74–75
  dividing into quadrants, 154–55
  drawing pictures in, 145–47
  expanding, 21
  moving, 21
  positioning text in, 85–87
  resizing, 21
  text organization, 84
  title bar, 61

P

PAINTRECT statement, 384
Paste command, 56–59
  copying text with, 58–59
PENMODE statement, 376–77

PENMODE statement *(continued)*
  BIC mode, 380
  Copy mode, 377
  OR mode, 378
  XOR mode, 378–80
PENNORMAL statement, 376
PENPAT statement, 369–70
PICTURE OFF statement, 315–18
PICTURE ON statement, 315–18
PICTURE$ function, 318
Pixels, 72
  BASIC statement used with, 87
  converting to binary numbers, 371–72, 375
  coordinates, moving text to, 361–62
  determining color of, 234
  erasing, 77
  locating, 73–75
  in mixing graphics with text, 354–58
  plotting, 75–78
    with QuickDraw pen, 366–67
  positions, tabbing to, 87
POINT function, 234
PRINT statement, 28, 30–32, 83
  and arithmetic expressions, 33–36
  with numbers, 33
  using commas in, 36–37
  using semicolons in, 38–40
PRINT# statement, 404–6
Printing
  program listing, 22
  screen dump, 360
Programs
  Animated Character Graphics, 165
  ASCII code, 210–11
  Chord Organ, 477–78
  Circle Drawing, 147
  copying parts of, 58–59
  creating, 47–48
  debugging, 24
  deleting text from, 23, 50–52
  duplicating parts of, 56
  Earnings, 411
  editing, 20, 49–59
  Effects, 466–67
  ending, 113
  Engine, 465

Programs *(continued)*
  event-trapping in, 228, 237–43, 465–66
  finding text in, 23
  Glissando, 463–64
  including remarks in, 112–13
  indexed branching, 172–74
  Investment, 127, 134
  lines, inserting new, 55
  listings, 22, 48
  loading, 22, 64–66, 67
  mixing text and graphics, 91
  Mouse Button, 135, 231
  Mouse Drawing, 145–47
  Mouse-Pointer Location, 137
  Moving-Sign, 208
  moving text in, 23, 56
  Musical Notes, 471
  naming, 60–61
  Number-Guessing, 149
  Pattern-Editor, 373–75
  Phone Book, 333–35
  Prelude in E Flat, 479
  Rectangle-Demo, 382–83
  replacing text in, 23, 53
  Rotating Space Buoy, 447, 450
  running, 48–49
  Salary, 183–86
  saving, 22, 59–64
    in compressed format, 63
    different versions, 64
    as protected, 63
  Select Rectangle, 231
  sending via modems, 63
  Siren, 464
  starting from desktop, 67
  Submarine, 156–57
  subprogram statements, 174
  subroutines, 160
    calling more than one, 172–74
    multiple, 162–64
    naming, 161
    nested, 170–73
  Textface, 163
  Tinkling, 464
  tracing loops in, 120–21
  transferring text between windows, 20

Programs *(continued)*
  Wave Shapes, 481
  Waveform Tester, 482–85
  windows, 310
  writing
    BASIC versions used in, 6–7
  Yankee Doodle, 470, 472
PSET statement, 75–78, 145–46
PTAB statement, 87, 166
PUT statement, 432–34
  action verbs
    AND, 440
    OR, 441
    PRESET, 439
    PSET, 438
    summary of, 442
    XOR, 438
  enlarging images with, 444–46
PUT# statement, 419, 424–25
  File Number parameter, 419
  Record Number parameter, 419, 424

Q

QuickDraw, 340–41
  accessing, 341–42
  ovals, 384–85
  parameters, 367
  pen, 362–65
    drawing patterns with, 368–70
    plotting pixels with, 366–67
    reference coordinates, 365
    restoring to default setting, 376
    size, 365–66
  rectangles, 381–86

R

Random access memory (RAM), 8, 368–70
Random numbers, 149–51
RANDOMIZE statement, 131
READ statement, 192–97
  creating menus using, 270–71
Relational operators, 118, 131–33
REM statement, 112–13
Reserved words, 30, 47
RETURN statement, 160–71
  dangling, 162
RIGHT$ function, 204–7
RND function, 150–51
RSET statement, 418
Run menu, 24, 55

S

Save command, 60
Screen. *See also* Pixels
  clearing, 33
  resolution, 79
Search menu, 23
Show List command, 47, 67
Sound
  duration, 42, 461
    changing, 465–66
  frequencies
    beat, 476
    changing, 463–66
    in musical scales, 466–72
    single, 465
    slides, 476
  frequency, 460–61
  glissando, 463–66
  harmonics, 476
  multivoice, 461, 473
    combining, 473–78
    program speed in, 473
  pitch, 42
  single-voice, 462–66
  volume, 461–62
  waveforms, 473–74
    creating, 478–86
    single-voice, 473
SOUND statement, 42, 161, 477
  Duration parameter, 461
  error message, 475
  Frequency parameter, 460–61
  Voice Number parameter, 461
  Volume parameter, 461
SOUND RESUME statement, 475
SOUND WAIT statement, 475
SPACE$ function, 223
SPC function, 223
Start command, 48, 55

Statements
  BACKPAT, 376
  BEEP, 41
  BUTTON, 282–88
  BUTTON CLOSE, 298–301
  CALL, 341–42
  CHR$, 350–52
  CIRCLE, 81–82, 124, 126
  CLOSE, 397–98
  CLS, 33
  DATA, 192–97
  DEFINT, 367
  DIALOG OFF, 322
  DIALOG ON, 322
  DIM, 180–81
  EDIT FIELD, 327
  EDIT MENU, 326
  END, 113
  ERASE, 191
  ERASERECT, 384
  FIELD, 416–17
  FILLRECT, 383
  FRAMERECT, 381
  GET, 432–34
  GET#, 421
  GOSUB, 160–71
  GOTO, 118, 138–40
  IF...THEN, 144–48
  IF...THEN...ELSE, 150
  INPUT, 100–105
  INPUT#, 399–400
  INVERTRECT, 383–84
  LINE, 78–81, 145
  LINE INPUT, 218–19
  LOCATE, 85–87
  LSET, 418
  MENU, 253–55
  MOUSE GOSUB, 237
  MOUSE OFF, 240–41
  MOUSE ON, 240
  MOUSE(0), 228–29, 231–32
  MOVETO, 361–62
  ON...GOSUB, 172–74, 262–63
  ON...GOTO, 262–63, 265–66
  ON MOUSE GOSUB, 240
  OPEN, 399–400
  OPTION BASE, 190–91, 448
  PAINTRECT, 384
  PENMODE, 376–77
  PENNORMAL, 376
  PENPAT, 369–70
Statements *(continued)*
  PICTURE OFF, 315–18
  PICTURE ON, 315–18
  PRINT, 28, 30–32, 83
  PRINT#, 404–6
  PSET, 75–78, 145–46
  PTAB, 87, 166
  PUT, 432–34
  PUT#, 419, 424–25
  RANDOMIZE, 131
  READ, 192–97
  REM, 112–13
  RETURN, 160–71
  RSET, 418
  SOUND, 42, 161, 477
  SOUND RESUME, 475
  SOUND WAIT, 475
  SWAP, 222
  TEXTFACE, 88–89, 162–64, 358–60
  TEXTFONT, 343–44
  TEXTMODE, 353
  TEXTSIZE, 89–90
  VARPTR, 370
  WAVE, 478
  WIDTH, 37, 83
  WINDOW, 306–9
  WRITE#, 397
Step command, 120–21
STR$ function, 14–15
STRING$ function, 222–23
Strings
  changing, 213
  comparing, 219
  concatenating, 203–4
  constants, 110
  converting to ASCII codes, 214–15
  parts of, 204–7
  searching for, 215–18
  sorting alphabetically, 219–21
  structure of, 202–3
  variables, 109–11
Subroutines. *See* Programs
Subtraction, 34
SWAP statement, 222

T

TAB statement, 83–85
Text
  character positions, 84–85

Text *(continued)*
  copying, 58
  deleting, 23, 51–53, 56
  deselecting, 52
  faces, 88–89, 358–60
  files, 63
  fonts, 343–50
  inserting new lines of, 55
  modes, 353–58
  moving, 23, 56–58, 361–62
  pasting, 57–58
  positioning, 85–87
  proportionally spaced, 84–85
  replacing, 23, 53
  selecting, 51–54
  sizes, 89–90
  styles, 88–89
    combined, 89
  transferring between windows, 20
TEXTFACE statement, 88–89, 162–64, 358–60
TEXTFONT statement, 343–44
TEXTMODE statement, 353
  BIC mode, 357–58
  Copy mode, 354
  OR mode, 355–56
  XOR mode, 356–57
TEXTSIZE statement, 89–90
TIME$ function, 40–41
Trace On command, 121, 125

U

UCASE$ function, 221

V

VAL function, 214–15
Variables, 95–114. *See also* Arrays
  arithmetic, 107–9
  buffer, 416–18, 422
  counter, 119–22
  horizontal position, 104–6, 124
  memory addresses of, 370
  multiple, 104–5
  names, 105–6
    group, 187
  simple versus array, 180–82
  string, 109–11
  string-array, 185–86

Variables *(continued)*
  subscripts
    referencing arrays with, 181–82
  values, 98–99, 108
    exchanging two, 222
  vertical position, 104–6, 124–25
VARPTR statement, 370

W

WAVE statement, 478
  Phase parameter, 474, 479–80
  Voice Number parameter, 474
  Wave Shape parameter, 474, 479
WHILE...WEND loop, 128–34, 145–46, 150
  logical expressions in, 129–31
  logical operators in, 153–54
  using relational operators in, 132–33
WIDTH statement, 37, 83
WINDOW statement, 306–9
  ID parameter, 306
  Rectangle parameter, 307–8
  Title parameter, 306–7
  Type parameter, 309
Windows
  absolute coordinates in, 309
  activating, 20, 24
  active, management of, 323–26
  arranging, 21
  closing, 8
  Command, 17, 20
  definition, 6
  as dialog boxes, 326–32
  document, 312–14
  List, 17
  moving, 312–14
  multiple, 310–12
  opening, 8, 24
  Output, 17, 28
  position of, 309–10
  queuing dialog events, 321–22
  refreshing, 314–21
  relative coordinates in, 309–11
  transferring text between, 20
  types of, 304–6
Windows menu, 24
WRITE# statement
  Expression List parameter, 397
  File Number parameter, 397

ABOUT THE AUTHORS

Mitchell Waite has been active in the personal computer industry since 1976, when he bought one of the very first Apple I computers from Steven Jobs. For the past several years his company, The Waite Group, has published dozens of award-winning computer books used throughout the world, including the best-sellers *Unix Primer Plus, C Primer Plus,* and *Assembly Language Primer for the IBM PC.*

Robert Lafore, co-author of the best-seller *Assembly Language Primer for the IBM PC,* is editorial director of The Waite Group. Robert has worked in the computer industry since 1965. He has also been a petroleum engineer in Southeast Asia, a newspaper columnist, and the head of his own software company.

Dr. Ira Lansing is instructor of Mathematics and Computer Science at the College of Marin, where he teaches courses in BASIC and C programming languages. Dr. Lansing has also been a consultant in the development of the instructional use of computers in the classroom.

THE MANUSCRIPT FOR THIS BOOK
was prepared and submitted to Microsoft Press
in electronic form. Text files were processed and
formatted using Microsoft Word.

Cover design and interior calligraphy
by Tim Girvin.
Interior text design by Karen-Lynne de Robinson.
The high-resolution screen displays were created on the
Apple Macintosh and printed on
the Hewlett-Packard LaserJet.

Text composition by Microsoft Press in Rotation
with display in Times Roman, using the CCI composition
system and the Mergenthaler Linotron 202
digital phototypesetter.

Cover art separated by Color Tech,
Redwood City, California. Printed on 12 pt. Carolina
by Strine Printing Company, York, Pennsylvania.
Text stock, 60 lb. Glatfelter Offset, supplied by
Unisource. Book printed and bound by Fairfield
Graphics, Fairfield, Pennsylvania.

OTHER TITLES FROM MICROSOFT PRESS

THE APPLE ENVIRONMENT

The Apple Macintosh Book
Cary Lu ISBN 0-914845-00-4 $18.95

MacWork MacPlay
Creative Ideas for Fun and Profit on Your Apple Macintosh
Lon Poole ISBN 0-914845-22-5 $18.95

Microsoft Multiplan: Of Mice and Menus
Models for Managing Your Business with the Apple Macintosh
The Waite Group, Bill Bono, and Ken Kalkis ISBN 0-914845-33-0 $16.95

Inside MacPaint
Sailing Through the Sea of FatBits on a Single-Pixel Raft
Jeffrey S. Young ISBN 0-914845-31-4 $18.95

AppleWorks
Boosting Your Business with Integrated Software
Charles Rubin ISBN 0-914845-47-0 $16.95

The Endless Apple
How to Maintain State-of-the-Art Performance on Your Apple II and IIe
Charles Rubin ISBN 0-914845-27-6 $15.95

GENERAL BOOKS

Online
A Guide to America's Leading Information Services
Steve Lambert ISBN 0-914845-35-7 $19.95

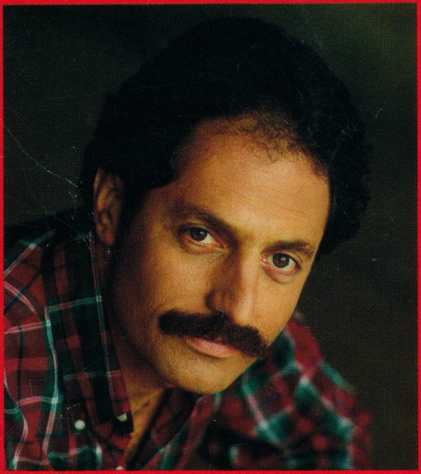
Out of the Inner Circle
A Hacker's Guide to Computer Security
"The Cracker" (Bill Landreth) ISBN 0-914845-36-5 $9.95 softcover
ISBN 0-914845-45-4 $19.95 hardcover

A Much, Much Better World
Eldon Dedini ISBN 0-914845-50-0 $6.95

Silicon Valley Guide to Financial Success in Software
Daniel Remer, Paul Remer, and Robert Dunaway ISBN 0-914845-09-8 $19.95

Learning Commodore 64 Logo Together
An Activity Book for Creative Parents, Teachers, and Kids
Kenneth P. Goldberg ISBN 0-914845-24-1 $14.95

Available wherever fine books are sold, or write to:
Marketing Department, Microsoft Press · 10700 Northup Way, Box 97200
Bellevue, Washington 98009



For every programmer captivated by the Macintosh—its high-resolution graphics, revolutionary user-interface, and its famous mouse—here is a lively and authoritative reference to its most popular and feature-packed language:

MICROSOFT BASIC

For newcomers to programming, MICROSOFT MACINATIONS provides hands-on tutorials on the fundamentals of Microsoft BASIC for the Macintosh. And for experienced programmers, MICROSOFT MACINATIONS is the definitive source of in-depth information on the Macintosh's unique characteristics—its menus, windows, buttons, QuickDraw graphics routines, as well as its exciting capabilities for sophisticated event-trap programming, graphics animation, and multi-channel sound generation.

Written with a sharp focus on programming concepts and techniques, MICROSOFT MACINATIONS is packed with program examples that are inventive and amazing, as well as instructive. These include a simple database program, a sound-generation program that synthesizes a four-voice Bach concerto, a clever pattern-maker program, and a fast-moving animation program.

The Waite Group, headed by the noted Mitchell Waite, has been producing best-selling computer books since the early days of microcomputing. The close working relationship between The Waite Group and Microsoft makes MICROSOFT MACINATIONS *the* source for comprehensive information on Microsoft BASIC.

$19.95

ISBN 0-914845-34-9